Norwegian University of
Science and Technology

System Scenario Based Application Mapping

Solveig Fure

Master of Science in Electronics
Submission date: December 2015
Supervisor: Per Gunnar Kjeldsberg, IET

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Description of Master Thesis

Candidate name: Solveig Fure

Title: System Scenario Based Application Mapping

In system scenario based design dynamic application behavior is exploited to optimize the application's execution costs (e.g., performance and energy consumption). Through code inspection and profiling, different dynamic scenarios are identified and optimized at design-time. Furthermore, techniques for scenario prediction and switching are developed. At run-time the application and platform are reconfigured according to the dynamically predicted scenario.

This master assignment is based on a project assignment in the previous semester that included a literature study of system scenario based design and experiments with two dynamic applications using a model developed for dynamic voltage and frequency scaling. This work shall be continued with a description of a general methodology for application profiling for detection of dynamic application behavior and its cause. The methodology shall be applied to selected codes from the SPEC CPU 2006 benchmark suite and possibly additional applications.

Co-supervisor: Yahya H. Yassin

Supervisor: Per Gunnar Kjeldsberg

Abstract

During the past decade, the processing requirements of embedded real-time systems have become more and more challenging; not only in terms of performance, but also regarding the energy efficiency. The dynamic nature of many of these systems has been recognized as an important feature to take advantage of. Scenario-based design is a well-known design methodology dealing with this. Identifying different use-case scenarios to optimize for is a strategy that has been used for a long time. Lately, another kind of scenario-based strategy has emerged; so-called *System scenarios*. Instead of classifying the behavior of a system through several use-cases, the actual costs of the system is considered, like resource usage or quality requirements. By thoroughly characterizing and analyzing the system behavior at design-time, the different system scenarios can be recognized at run-time without much overhead. The system can then be tailored to the identified scenario by employing the optimal settings that have been determined at design-time. It must however be enough exploitable dynamism present in the application to make up for the system scenario overhead.

The system scenario research field has well described theoretical design methodologies, even considering multiple cost dimensions. It can however be a long distance between the theoretical methodologies and an actual design process, especially when the dynamic cost is multi-dimensional. There is a considerable increase in the design complexity from adding just one more cost dimension to the design, motivating the need for a precise and concrete methodology. This thesis presents methods for detecting and exploiting dynamism in applications through profiling and code inspection, and the results from applying these methods on four applications, where three of them are a subset of the SPEC CPU™ 2006 integer benchmarks. The focus is on exploring both memory and CPU dynamism. Identifying the dynamism in an application can become very complex when considering multi-dimensional dynamism. An extension to the system scenario methodology which addresses this problem is therefore suggested. The scenario identification step of the original methodology is split into several smaller steps and executed separately for each of the considered dynamisms. This way the designer does not have to keep the entire design space in mind, but can focus on each kind of dynamism separately. *Sub-scenarios* are identified for each kind of dynamism, and then combined to form a scenario set where the total dynamism is exploited. The methodology is targeted at simple single-core platforms.

Several papers present promising results of the system scenario methodology when only considering CPU optimizations such as Dynamic Voltage and Frequency Scaling (DVFS). For applications requiring much memory however, DVFS can become infeasible as it means longer stand-by time for the memory. Memory typically contributes significantly to the total energy consumption, which motivates the recent introduction of memory

system scenarios with dynamically reconfigurable memories. In this work, the dynamism in applications that are *both* memory and computationally intensive are exploited by combining DVFS and memory reconfigurations. Much CPU and memory dynamism is identified in the considered benchmark applications, and up to 31% of the total energy is saved by applying the modified system scenario methodology. Up to 47% is saved for some of the low-workload situations. Characteristics of an Intel Pentium processor and CACTI memory models are used to estimate the energy savings.

Sammendrag

I løpet av de siste tiårene har kravene som stilles til innvevde systemer blitt stadig mer utfordrende, både hva angår ytelse og energieffektivitet. Mange av disse systemene har imidlertid dynamiske egenskaper som kan utnyttes. Den scenario-baserte design metodikken er et velkjent hjelpemiddel for å håndtere dette. Den tradisjonelle varianten går ut på å klassifisere oppførselen til et system gjennom såkalte bruker-scenarioer, for å deretter kunne optimalisere systemet avhengig av behov. Senere har en annen variant, med såkalte *system scenarioer*, begynt å få oppmerksomhet. I stedet for å dele inn i scenarioer basert på ulik oppførsel, ser man på variasjoner i den faktiske kostnaden til systemet. I løpet av design-fasen karakteriserer og analyserer man systemet grundig, slik at de ulike scenarioene som skulle oppstå i det ferdige systemet (altså i bruks-fasen) kan kjennes igjen uten mye ekstraarbeid. I design-fasen finner man også de optimale konfigurasjonene for hvert scenario. Det ferdige systemet kan dermed i bruks-fasen enkelt tilpasses scenarioene som kjennes igjen ved å ta i bruk disse forhåndsbestemte konfigurasjonene. Denne fordelingen av kontroll mellom design-fasen og bruks-fasen er sentral for system scenario-metodikken.

Forskningen på system scenarioer har resultert i veldefinerte teoretiske designmetodikker som skal fungere godt selv for å utnytte dynamikk i flere dimensjoner. Avstanden mellom en teoretisk metodikk og den faktiske design prosessen kan imidlertid være stor, da spesielt hvis flere former for dynamikk skal utnyttes samtidig. Kompleksiteten i designprosessen øker betydelig bare man tar én ekstra dynamisk kostnad med i betraktningen. En presis og konkret metodikk kan i så fall forenkle designprosessen. I denne rapporten presenteres metoder for å karakterisere og analysere to ulike former for dynamikk, nærmere bestemt i CPU- og minnebruk, samt en optimaliseringsteknikk for å utnytte dynamikken best mulig med en gitt platform. Dette testes og demonstreres på et utvalg applikasjoner, hvorav tre er hentet fra SPEC CPU™ 2006. Det blir fort mye å tenke på dersom man skal identifisere og karakterisere flere former for dynamikk samtidig. For å redusere dette problemet foreslås det derfor her en utvidelse av system scenario metodikken. Forslaget går ut på å splitte opp identifikasjonsprosessen i flere mindre steg for hver av dynamikk-dimensjonene som betraktes. Slik trenger ikke designeren å tenke på alle dimensjonene samtidig, men kan fokusere på hver dynamikk for seg. Under-scenarioer som utnytter hver form for dynamikk enkeltvis identifiseres, og blir deretter kombinert til et sett med scenarioer som utnytter den totale dynamikken. Metodikken er tiltenkt enkle énkjernes-platformer. Karakteristikken til en Intel Pentium prosessor samt minnemodeller fra CACTI brukes for å estimere energibesparelsene metodikken gir for nevnte applikasjoner.

I flere publikasjoner er det blitt vist lovende resultater for system scenario-metodikken, men det er ofte kun CPU-optimaliseringer som benyttes, f.eks. DVFS (dynamisk spenning- og frekvensskalering). Dersom en applikasjon krever mye minne vil ikke alltid DVFS alene gi gode resultater, siden det innebærer lengre standby-tid for minnet. Siden energifor-

bruket i minnet kan utgjøre en stor del av det totale energiforbruket har dynamisk konfigurerbart minne nylig blitt introdusert i system scenario-sammenheng. I dette arbeidet kombineres DVFS med ulike konfigurasjoner av dynamisk minne for å kunne utnytte dynamikken i applikasjoner som er både CPU- og minneintensive. Det identifiseres mye dynamikk i utvalget applikasjoner, både i CPU- og minnebruk, hvorav mye kan utnyttes. Opptil 31% av den totale energien spares i implementasjoner som designes vha. den modifiserte system scenario-metodikken. Opptil 47% spares i de mindre krevende situasjonene som oppstår i applikasjonene.

Preface

This report is the result of a master thesis work conducted through the fall of 2015. It concludes my Master of Science degree in Electronics, at the Department of Electronics and Telecommunications, NTNU. Some parts of this report are taken from a system scenario project conducted during the spring of 2015. More specifically; the first part of the abstract and introduction, the theory chapters on system scenario development and DVFS, and the developed prime number application.

The focus of the master thesis is slightly different from the project work. Whereas I in my project work used a quite platform-directed approach, this thesis is more general in terms of hardware. The prime number application studied in the project work is studied further in this thesis. Additionally, three SPEC2006 benchmark applications are selected for system scenario design. The RTS identification techniques used in the project work had to be changed somewhat in order to design system scenarios without being able to test on a platform. Other applications also required a more general approach. Much work has been put down to make the benchmarks run on my premises, getting an understanding of how they work and whether or not the dynamism could be taken advantage of. Additionally, dynamic memories are introduced as a system knob

I would like to thank my supervisor Per Gunnar Kjeldsberg for all the guidance throughout this work. He introduced me to the concept of System Scenarios, which immediately caught my interest, and then suggested this project after considering my background and experience. Choosing to move on with the same topic and supervisor for my master thesis was an easy decision that I have not regretted. Thanks for all input and ideas, both regarding the theory and academics, the structure of the master work, and the report formalities.

I would also like to thank my co-supervisor Yayha H. Yassin. We had some very useful discussions where he shared his findings with me and evaluated my ideas.

Disclaimer: The SPEC performance results in this paper do not comply with the SPEC reporting rules [1].

Contents

Description of Master Thesis **i**

Abstract **iii**

Sammendrag **v**

Preface **vii**

List of abbreviations **xiii**

1 Introduction **1**
1.1 Contribution 3
1.2 Report structure 4

2 Theory and previous work **5**
2.1 System scenario design methodology 5
2.1.1 Design-time 5
2.1.2 Run-time 7
2.1.3 Cost 8
2.1.4 Memory-aware system scenarios 9
2.1.5 Task Concurrency Management (TCM) 9
2.2 Scaling frequency and voltage 11
2.2.1 Power and energy 11
2.2.2 Without static power 12
2.2.3 With static power 13
2.3 Dynamic Voltage and Frequency Scaling 13
2.4 Memory organizations 14
2.5 SPEC CPU 2006 16
2.6 Other related work 16

3 Platform and profilers **19**
3.1 Intel Pentium M 19
3.1.1 Power estimations 20
3.2 Memory Platform 23
3.3 Scenario Manager 24
3.3.1 Software implementation 25
3.3.2 Hardware implementation 25

3.4 Profilers . . . 27
3.4.1 Zoom . . . 27
3.4.2 Valgrind Massif . . . 27
3.4.3 Valgrind Cachegrind . . . 28

4 Applied methodology **29**
4.1 System requirements and characteristics . . . 31
4.2 Scenario identification . . . 31
4.2.1 DVFS dynamism . . . 32
4.2.2 Memory dynamism . . . 34
4.2.3 Scenario identification summary . . . 36
4.3 Exploration and exploitation . . . 38
4.4 Prediction . . . 42
4.5 Switching . . . 42
4.6 Theoretical evaluation of the methodology . . . 42

5 Applications **45**
5.1 Prime number checker . . . 46
5.1.1 DVFS dynamism . . . 47
5.1.2 Memory dynamism . . . 49
5.1.3 Exploration and Exploitation . . . 49
5.1.4 Prediction and Switching . . . 49
5.1.5 Results . . . 51
5.1.6 Changed assumptions . . . 51
5.2 456.hmmer . . . 52
5.2.1 DVFS dynamism . . . 52
5.2.2 Memory dynamism . . . 57
5.2.3 Exploration and Exploitation . . . 59
5.2.4 Prediction and Switching . . . 62
5.2.5 Results . . . 62
5.2.6 Other observations . . . 63
5.3 429.mcf . . . 65
5.3.1 DVFS dynamism . . . 66
5.3.2 Memory dynamism . . . 70
5.3.3 Exploration and Exploitation . . . 71
5.3.4 Prediction and Switching . . . 72
5.3.5 Results . . . 76
5.3.6 Other observations . . . 76
5.4 464.h264ref . . . 79
5.4.1 DVFS dynamism . . . 79
5.4.2 Memory dynamism . . . 82
5.4.3 Exploration and Exploitation . . . 84
5.4.4 Prediction and Switching . . . 86
5.4.5 Results . . . 88

6 Discussion and Future Work **91**
6.1 Applications . . . 91
6.2 Memory related . . . 92
6.3 CPU related . . . 94
6.4 Methodology . . . 94

7 Conclusion **97**

Appendix **103**

A Miscellaneous **105**
A.1 Dynamism in memory accesses . . . 105
A.2 Prime number probabilities . . . 106
A.3 Controlling the benchmarks . . . 107
A.3.1 How to run HMMER . . . 107
A.3.2 How to run MCF . . . 107
A.3.3 How to run H.264 . . . 108
A.4 MCF data set generator . . . 108
A.5 Viterbi algorithm . . . 108
A.6 CACTI . . . 108

List of abbreviations

IoT Internet of Things 1

RTS Run Time Situation 5

TCM Task Concurrency Management 9

TN Thread Node 9

TF Thread Frame 9

DVFS Dynamic Voltage and Frequency Scaling 11

DFS Dynamic Frequency Scaling 13

DVS Dynamic Voltage Scaling 13

NUMA Non-Uniform Memory Access 15

TDP Thermal Design Power 20

HMM Hidden Markov Model 52

MCF Minimum Cost Flow 65

CPI Cycles Per Instruction 76

Chapter 1

Introduction

Energy efficiency is becoming increasingly important in a wide range of computer systems. For data centres and servers, the significant cost of power for computation and cooling is a big motivation to reduce energy [2]. At the other extreme, the fast developments within mobile battery-powered devices require more and more performance with a very limited battery-capacity [3]. The vast opportunities with future Internet of Things (IoT) developments make energy efficiency even more relevant. Power management is therefore a critical consideration when designing many of today's computer systems. As a result, there are several features available in modern hardware that can be used to reduce power. Efficient and smart software control is however required to take full advantage of this potential, and power management has thus been recognized as an important research field.[4]

Maintaining the reliability of an embedded system while saving energy is a big challenge, especially with the ever increasing complexity of both the applications and the platforms on which they run. Many embedded system applications have dynamic performance requirements. When a system cannot adapt its configurations to different workloads at run-time it means that it must be designed to always accommodate for the worst-case scenario. This leads to expensive systems that use a lot more energy than necessary much of the time. On the other hand, if the system is able to adapt its configurations at run-time to the current workload, there is a big potential to reduce the energy consumption. Signal processing systems often have dynamic processing requirements that should be exploited, e.g. multimedia decoders like JPEG and H.264. Especially when decoders are placed in handheld devices and the energy budget is limited, smart power management is important. These applications are typically dynamic due to for-loop structures with input-dependent ending conditions, resulting in a number of iterations that is unknown at design-time. Much energy can however be saved if one is able to take advantage of this dynamism; the worst case workload can easily be as much as ten times the best case workload in a modern streaming application.[5]

Creating optimal power management mechanisms that take advantage of such dynamic behavior is not a trivial task. Often, sufficient information is not available at design-time, and completely postponing the decision until run-time gives too much run-time overhead. The *System Scenario* design [6] is introduced as a methodology to take better advantage of the dynamic nature of embedded system applications, while keeping the necessary overhead at an acceptable level. An important feature of this methodology

is that the workload analysis and system optimization is performed *both* at design-time and run-time. At design-time, the system's execution is divided into possible system scenarios, representing typical workload behaviors, based on cost analysis and profiling. Optimal system configurations are then found for each scenario. At run-time, when a particular scenario is recognized, the corresponding system configurations are applied. The system can thus be optimized at run-time with an acceptable overhead. A more detailed explanation of the methodology is given in Chapter 2.

The run-time configurability of the system is characterized by a set of *system knobs*. Dynamic Voltage and Frequency Scaling (DVFS), introduced in 1990 [7], is the most common platform configuration used within the system scenario methodology [8]. By decreasing the clock frequency of a processor and thus enabling a corresponding down-scale in the supply voltage, the power consumption can be reduced when performance requirements are low. The later developments with smaller feature sizes, lower operating voltages, more leakage current, and better sleep modes have however made DVFS by itself less useful, and the effects more difficult to model [9]. Furthermore, as pointed out by Snowdon et al. [10], the huge range of different applications makes the power and performance prediction difficult to generalize. Changing frequency leads to changes in power and performance that depend on the exact instruction mix of the application. For a simple application, the exact behavior on a given platform can be characterized at design-time [11]. If the application is more complex and/or depends on dynamic input data, this might be difficult or even impossible. This is where the system scenario methodology comes in.

There are many other possible system knobs, e.g. memory-related configurations as suggested by Filippopoulos et al. in [12]. In [13] they present an application-independent memory-aware system scenario methodology to exploit dynamic memory requirements. Memory energy is reduced significantly by adapting run-time reconfigurable memory banks according to the predicted scenarios. The future developments within reconfigurable platforms can provide an even better foundation for system scenarios. Heterogeneous cores and customization through for example accelerators and dynamically configurable logic have been recognized as promising ways of improving energy efficiency. Multiple cores and customization are predicted to be the main drivers for microprocessor performance in the future [4]. A reconfigurable cache architecture is suggested in [14], concluding that dynamically changing write-strategy and replacement-strategy are feasible techniques. It is reasonable to believe that the system scenario design methodology will be used to exploit multi-dimensional dynamism, i.e. using multiple system knobs concurrently. The result can easily be a sub-optimal system knob setting unless a thorough and well-structured scenario exploration has been carried out. E.g., when DVFS and dynamic memory are used as system knobs, the feasibility of a DVFS setting depends on the current memory setting, and vice versa. The complexity of such designs makes it necessary with a precise and concrete methodology.

Finding suitable applications to evaluate such a methodology can be a challenge by itself. The SPEC benchmark suites contains a variety of the most important types of program behavior, and has become an industry standard for measuring processor and compiler performance. SPEC CPU™ 2006 [15] is SPEC's newest benchmark suite, composed of a wide range of different benchmark tests and inputs that are all based on actual, relevant applications [16]. The benchmarks are characterized by large input datasets and long

execution times (compared to CPU2000). Given that there is some exploitable dynamism present in these applications, they should provide a suitable measure for evaluating the system scenario methodology.

1.1 Contribution

The intention of this work is to analyze the dynamism in a set of applications, develop system scenario implementations where the dynamism is exploited, and based on the gathered experiences suggest modifications or extensions to the system scenario methodology.

System scenarios represents a quite new field of research, so much of the previous work investigates system scenarios with a very theoretical approach. Additionally, most previous work is focused on scenario prediction based on control variables. Data variable based prediction is however necessary in many dynamic systems. In this work, the dynamic CPU and memory requirements are found for a set of applications through profiling and code inspection. Both data and control variables that cause dynamism are recognized. The considered applications are three of the SPEC2006 integer benchmarks, in addition to a prime number detection application. The procedure for detecting CPU and memory dynamism is somewhat generalized and suggested as an extension to the original system scenario identification step. A slightly modified version of the traditional system scenario methodology is further used to develop system scenario implementations. The modifications come as a consequence of the dynamism to be exploited and the nature of the applications.

The traditional system scenario methodology is well described for one-dimensional dynamism [6]. The complexity of identifying the scenarios and optimizing the platform does however increase drastically when adding more dimensions to the cost function. E.g., the memory energy consumption modelled in [13] depends on the execution time, so that introducing DVFS into the memory scenario design will also affect the memory energy. This report presents a methodology that is closely related to the traditional methodology [6], but with some modifications in order to better deal with multi-dimensional dynamism. The methodology is developed and presented using DVFS and dynamically reconfigurable memory, but other system knobs can also be used.

The whole process is described, from identifying the benchmarks that are the most promising candidates for system scenario development, to the actual results of such scenario implementations. The objective with the implementations is to achieve significant reductions in energy consumption through dynamic run-time reconfiguration of the platform, while preserving the performance requirements (deadlines). It is demonstrated how both data variables and control variables can be used to detect/predict scenarios, and how these variables can be found through profiling and code inspection. A considerable amount of the work is related to the design-time scenario exploration and exploitation steps, in which the optimal system settings are found.

The following list summarizes the contributions of this work:

- Evaluation of the suitability of Intel Pentium M for system scenario-based DVFS. Maximum CPU energy reduction through DVFS is estimated to 34%.
- Methods for finding CPU and memory dynamism using various profilers.

- Adaptations of the traditional system scenario design methodology in order to identify and exploit multi-dimensional dynamic behavior, targeting CPU and memory dynamism specifically.
- Algorithms for exploring and exploiting the possible platform configurations.
- Analysis of the CPU dynamism in a prime number checking application. The developed system scenario implementation reduces the CPU energy consumption with 20.2% compared to a static implementation.
- Delayed scenario prediction through loop unrolling to reduce the rate of scenario mispredictions. Demonstrated on the prime number checker.
- Analysis of the CPU and memory dynamism in three SPEC CPU 2006 applications. Suggested system scenario implementations obtain energy reductions up to 31.2%.

1.2 Report structure

The rest of the report is structured as follows: First, the results of a literature survey on system scenarios, DVFS and dynamic memory configurations are presented in Chapter 2. Chapter 3 describes the memory and CPU models of the assumed platform and briefly outlines the profilers that are used. Chapter 4 presents the general methodology that is used for identifying, exploring and exploiting dynamic behavior, with special focus on the deviations from the original methodology. Chapter 5 describes each of the applications that are studied and the system scenario design decisions and assumptions that are made. The resulting system scenario implementations with corresponding energy reductions are also presented in this chapter. Chapter 6 discusses the presented methods and results, and suggests future work. Finally, Chapter 7 concludes the report.

Chapter 2

Theory and previous work

2.1 System scenario design methodology

The design process for energy efficient dynamic systems can get quite complex, so following a general design methodology can be useful. In this section, the system scenario design methodology is presented thoroughly.

Gheorghita et al. defines system scenarios as follows[6]:

System scenarios group system behaviors that are similar from the multidimensional cost perspective, such as resource requirements, delay and energy consumption, in such a way that the system can be configured to exploit this cost similarity.

A dynamic embedded system typically has varying execution costs depending on the input and the state of the system. If the functions describing these variations are not known at design-time the system must be implemented based on worst case costs. By utilizing a scenario driven design process however, the dynamic properties of the embedded system can be identified and accounted for already in the the design-phase, which makes it possible to exploit the dynamism at run-time. This two-phased partitioning is a characteristic feature of the system scenario methodology.

The methodology and terminology presented in this section is based on work by Gheorghita et al.[5, 6].

2.1.1 Design-time

At design-time, the systems behavior is analyzed and classified from a possibly multi-dimensional cost perspective, that usually involves energy consumption and workload. The system's behavior is divided into Run Time Situations (RTSs) that can be recognized through run-time observable variables (RTS parameters). The system execution is thus a sequence of RTSs. A dynamic system typically has many different RTSs, and each of these can be identified and treated as units with associated costs. The number of RTSs can however be enormous even for simple systems, which is why RTSs with similar costs should be grouped together in a *System scenario*.

At run-time, the next scenario to be executed is predicted from the RTS parameters; a set of system parameters that can be both data and/or control-related. By recognizing a scenario *before* its execution, the system can be configured for optimal execution of this particular scenario. The optimization happens by adjusting so-called *system knobs*, which

are hardware or software settings that can be changed at run-time. Some examples are:

- Supply voltage
- Clock frequency
- Mapping to processing elements
- Memory organization
- Code transformations
- Executing different versions of the code (e.g. compiled with different optimization settings)

From a system specification to a final system, the design-time phase can be divided into the steps shown in Figure 2.1. Following is a description of each of these steps.

Figure 2.1: Overview of the system scenario methodology at design-time. Simplification of figure from [5, p.19].

Scenario identification

The identification step can be divided into RTS parameter discovery and RTS clustering. The RTSs are discovered through profiling and code inspection, and the relevant RTS parameters are selected. Extensive profiling may be necessary to discover how the dynamic nature of a complex application is best exploited. By evaluating the application's RTSs thoroughly at design-time, one can avoid spending unnecessary run-time resources on detecting dynamism. Scenarios are constructed by clustering the RTSs that have similar costs. The overhead costs and frequency of occurrence should be considered, to the extent that is possible at such an early stage.

Detecting *every* possible RTS at design-time might be impossible (or at least infeasible) for complex systems, which motivates the need for a backup scenario. This scenario will typically have high performance so that it can accommodate for any given situation. By implementing a backup scenario the system can give hard guarantees even for unforeseen behavior.

Prediction/detection

To be able to configure the system before a scenario occurs, a scenario predictor that can run along-side the application at run-time must be developed. The predictor will monitor the selected RTS parameters and from these determine the optimal future system scenario. The quality of this predictor must be traded off against the run-time prediction overhead. Should the prediction fail, the mentioned backup scenario can be used.

For systems where all possible RTSs have been identified and the prediction function is applied after all RTS parameters are known, the prediction accuracy will be 100%. This is called scenario *detection*. Often the scenarios will be detected too late when using this approach.

Developing a predictor/detector might influence the already determined scenario set, e.g. some of the scenarios may be too difficult to recognize. The scenario set should therefore be re-evaluated and maybe redefined. The energy consumption of the predictor/detector depends on the exact implementation. A few alternatives are somewhat discussed in Section 3.3.

Exploitation

After a predictor has been developed, the optimal application and platform settings that should be applied for each scenario must be found. A scenario should not be optimized completely independent of the rest of the scenario set, but with the scenario switching in mind. Otherwise, the system configurations for each scenario may become too different to even allow run-time switching. Several scenarios are typically combined in this step and the RTSs might even have to be completely remapped to the scenarios now that the actual configuration options are known.

Switching

Based on the obtained scenario prediction and exploitation, smart and effective mechanisms for switching between the scenario configurations must be developed. Switching to a better suited scenario may not always be a good idea. The gain of the scenario switch must be weighted to the associated costs of switching. The amount of time in which this new scenario will be optimal has much influence on the decision, as well as the necessary overhead it takes to do the switching. This can vary a lot depending on the actual configurations that must be made. E.g., frequency scaling, voltage scaling or changing the state of a memory bank (active, sleep, shut down) will all require different amounts of time and energy. It may again be necessary to reconsider the scenario set now that the switching overhead can be taken into account. Maybe some scenarios must be merged because of the associated switching costs.

Calibration

Some scenario systems also include a *Calibration* step. At design-time, a calibration routine is then developed, which can evaluate the current set of scenarios by observing how the system performs at run-time. In case some unforeseen behavior must be incorporated into the scenario set, the calibration routine can add, remove or change the system scenarios.

2.1.2 Run-time

At *Run-time*, the classification found at design-time is used to take advantage of the dynamism of the system. The identified set of RTS parameters are monitored so that the system scenario of the predicted RTS can be selected. The run-time phase can be

organized quite similarly to the design-time phase, as shown in Figure 2.2. Following is a description of each of these steps.

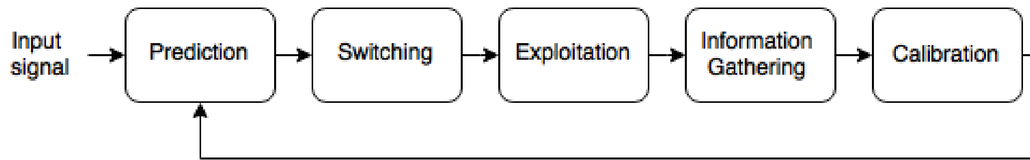


Figure 2.2: Overview of the system scenario methodology at run-time. Simplification of figure from [5, p.19].

Prediction

Based on actual RTS parameter values, a scenario is selected from the pre-determined scenario set. The parameter evaluation can be continuous, periodic or event-triggered.

Switching

If the predicted scenario is different from the current, and if the cost of switching to this scenario is lower than the benefit of this scenario, the switch should be performed.

Exploitation

At run-time, this step simply involves running the system with the optimal system scenario settings, so that the current state of the application is exploited.

Information Gathering and Calibration

In complex and very dynamic systems, a calibration routine can be used to further optimize the scenarios. The calibration should happen sporadically, based on run-time statistics found by information gathering mechanisms.

2.1.3 Cost

Within each scenario the system configurations are fixed, meaning that all RTSs in a given scenario will have the same expected cost. The resource usage of this scenario must be sufficient for the worst-case RTS of the scenario, which means that this worst-case RTS determines the expected cost of all the situations within its scenario. Usually, one or more of the following costs are considered:

- Processing cycles
- Energy consumption
- Memory accesses
- Heat production

- Expected quality of result
- Timing/deadlines and cost of missing deadlines

If the cost is a sum of several variables, it can be represented as a weighted sum or as an N-dimensional Pareto set. The latter is more difficult to implement, but might be advantageous with a multidimensional cost in order to avoid inconsistencies and suboptimal scenario sets [5, p.19].

2.1.4 Memory-aware system scenarios

Filippopoulos et al.[12, 13] proposes memory-awareness as an extension to the established system scenario methodology, by including memory costs in the design exploration. Using this extended methodology on an epileptic seizure predictor and a Viterbi encoder they are able to reduce the memory energy consumption with 40 to 70%[12]. The memory access pattern and data reuse size are some of the characteristics that are used to identify RTSs. In [13] the methodology is demonstrated on a set of multimedia benchmarks, resulting in a reduction of 35 to 55% in memory energy consumption. An extended memory model with state-of-the-art memories is presented and used for the scenario exploration. Different system knobs are suggested to be used in a memory-aware system scenario design methodology; e.g., mapping data to different memory banks, turning on and off memory banks, employing memory sleep modes, and tuning the operating frequency exactly to the access frequency needed.

2.1.5 Task Concurrency Management (TCM)

TCM is a methodology closely related to the system scenario methodology. TCM is developed by IMEC and its partners [17], addressing the problem of mapping dynamic and concurrent tasks on multiple processors for energy-constrained real-time embedded systems. To adapt the scheduling to the dynamism in the tasks, the scheduling is split into two phases; design-time and run-time scheduling. As already described, this two-phased strategy is also characteristic for the system scenario methodology. Following is a summary of the main TCM concepts, adapted from [17].

TCM Work flow

Four separate stages make up the TCM work flow; gray-box system model construction (system-level modelling), concurrency improving transformations (system model optimization), design-time scheduling and run-time scheduling. At the high-level specification, the application is represented as a set of Thread Frames (TFs). Each TF is partitioned into a set of Thread Nodes (TNs). These are the basic low-level scheduling units. The methodology then seeks to schedule the TNs onto a set of homogeneous or heterogeneous processors, both cost-optimal and constraint-driven.

Gray-box modelling

A Multi-Task Graph (MTG) model is created at design-time (first step). This is a two-level hierarchical task graph, specifying the application at a gray-box abstraction level.

The intention of the gray-box model is to hide irrelevant parts of the code, i.e. parts that are deterministic and independent of input or situation, making it easier to focus on the non-deterministic, dynamic parts. This technique can be very useful when identifying dynamic behavior in system scenario design if the application code is complex.

Neither the classic black-box or the white-box models are suited to represent an application during TCM. The abstraction level needs to be something in between; hence the gray-box model is used. The gray-box model has two abstraction levels, as shown in Figure 2.3. At the high-level, the model represents the application by dividing it into a set of TFs. These are typically functions (or several functions), and must not contain any non-deterministic external interactions. At the lower level, the gray-box model partitions the TFs into TNs, which are characterized by constant execution times and energy consumptions, and are therefore considered as black-boxes. The TFs however are considered as white-boxes, as all of their internal structures are visible in the model. The dynamic behavior inside these TFs can be captured in system scenarios. The hierarchical combination of white- and black-boxes makes up the gray-box model [17].

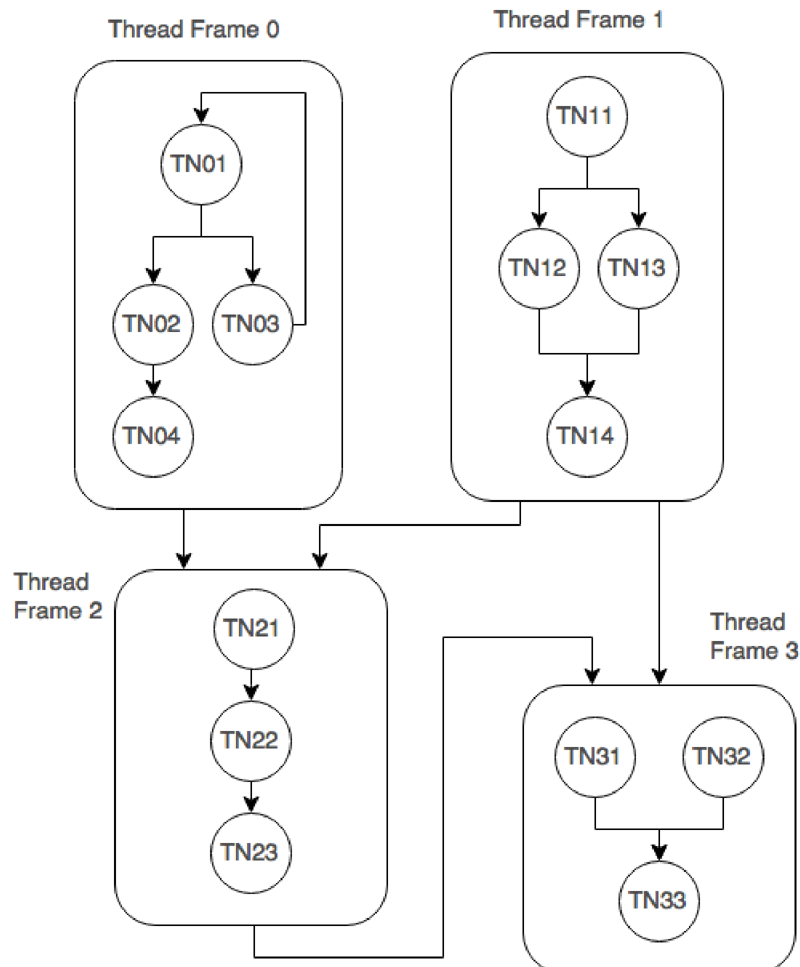


Figure 2.3: Example of how the thread frames and thread nodes can be organized in a gray-box model. Simplification of figure from [17, p.50].

According to [17], TFs and TNs are typically identified as follows: All dynamic task creation, explicit synchronization and external trigger events are identified. These are the

boundaries of the TFs, and so the TFs are found between these events, typically wrapped by functions. The TFs should be maximally sized pieces of functionality. To identify TNs, some insights of how the application works is necessary. A TN is a maximal set of connected operations where the worst-case behavior and actual behavior are similar enough, giving a somewhat deterministic execution latency. Deciding the TN granularity is however a complex task, and the reader is referred to [17] for a more detailed description.

Two-phase scheduling

Pareto curves of the TFs and their Pareto-optimal schedules of TNs are generated at design-time, and the run-time scheduler simply dispatches specific schedules instead of generating schedules on the fly. This is the essence of two-phase scheduling. The run-time scheduling will happen at the granularity of TFs. When a new TF is initiated because of external events, the scheduler selects active scenarios from the input data, and schedules them so that deadlines are reached and energy consumption is within a constraint[17, p. 48]. By separating the design-time and run-time scheduling the system becomes more flexible at run-time, and can to some extent accommodate for unforeseen timing demands. By selecting the optimal combination of Pareto points the energy cost of a specific timing constraint usually spanning multiple TFs can be minimized. The computation complexity and overhead at run-time are also minimized.

2.2 Scaling frequency and voltage

The next section describes dynamic frequency and voltage scaling and thus gives a theoretical foundation and motivation for some of the system configurations performed. Before going into the theory of Dynamic Voltage and Frequency Scaling (DVFS), some important power and energy equations are presented in the current section. These equations will also be used in Section 3.1.1 to create a CPU power model.

2.2.1 Power and energy

The power dissipated in a two-terminal element at any time is given by the product of the voltage $v(t)$ across its terminals and the current $i(t)$ running through it: $p(t) = v(t) \cdot i(t)$ [18, p. 16, p.27, p.595]. Correspondingly, the average power consumption P of an electronic system is given by Equation 2.1, where I is the average current and V is the supply voltage.

$$P = I \cdot V \tag{2.1}$$

Power is by definition the rate at which energy is consumed [18, p. 17], so by multiplying the average power P with some time interval T, the total energy is found:

$$E = P \cdot T \tag{2.2}$$

Total power P dissipated per time by an integrated circuit can be expressed as the sum of static power P_{static} and dynamic power $P_{dynamic}$ [19], as shown in Equation 2.3.

$$P = P_{dynamic} + P_{static} \tag{2.3}$$

Static power is consumed whenever the circuit is powered, regardless of circuit activity, while the dynamic power results from charging and discharging of capacitors [18, p.603]. In logic gates, the static power dissipation P_{static} is proportional to V^2, according to the simplification given in [18, p. 604]. Static power is also proportional to the leakage current I_{leak}:

$$P_{static} = V \cdot I_{leak} \tag{2.4}$$

I_{leak} is the current that leaks through transistors that are switched off. It exists whenever the chip is powered, but is independent of frequency [20]. The leakage is mainly set by the sub-threshold current of the transistor, which depends on threshold voltage, channel physical dimensions, V_{DD} and many other factors [21]. Leakage current is rather complex to model, and is not investigated further here.

Dynamic power on the other hand is more straightforward, and can be expressed as shown in Equation 2.5[19, 18].

$$P_{dynamic} = V \cdot I_{dynamic} = A \cdot C \cdot V^2 \cdot f \tag{2.5}$$

A is the activity factor, C is transistor gate capacitance, V is the supply voltage and f is the operating frequency. As the voltage is squared, reducing this factor leads to a significant dynamic power reduction. Dynamic power scales linearly with frequency.

By combining Equation 2.4 and Equation 2.5 and multiplying with a time interval T, we can calculate the energy consumption of a chip as shown in Equation 2.6 and Equation 2.7.

$$E = P \cdot T = (P_{dynamic} + P_{static}) \cdot T = (AC \cdot V^2 \cdot f + V \cdot I_{static}) \cdot T \tag{2.6}$$

$$E = P \cdot T = V \cdot I \cdot T = V \cdot (I_{dynamic} + I_{static}) \cdot T \tag{2.7}$$

DVFS has the potential to reduce both static and dynamic power dissipation as both sizes depend on voltage and/or frequency.

2.2.2 Without static power

By assuming that the static power is very small relative to the dynamic power (which can be true for high frequency operation [19]), the energy equations can be simplified:

$$E = P \cdot T = P_{dynamic} \cdot T = AC \cdot V^2 f \cdot T \tag{2.8}$$

The frequency can be written as $f = cycles/T$, where $cycles$ is the number of clock cycles used during time T. This gives:

$$E = AC \cdot V^2 \cdot cycles \tag{2.9}$$

Note that this equation is independent of time and frequency.

2.2.3 With static power

The potential energy savings from DVFS are not necessarily that straightforward. Recent developments in processor and memory technologies have made the ratio of dynamic power to static power smaller, so that optimizing voltage and frequency to reduce the dynamical power consumption makes less overall impact. Processor clock frequencies are more saturated, sleep modes are improving, and the dynamic power range is decreasing[9]. All of these developments limit the potential power reductions from DVFS. Experimental results also show that the gains from DVFS are smaller than predicted by simplistic power models like that in Equation 2.9 [19, 9].

By including the power consumption due to leakage current (Equation 2.4), the energy calculations get a lot more complicated, and the time T spent on a task has to be taken into account.

2.3 Dynamic Voltage and Frequency Scaling

DVFS is a well-known technique for reducing energy consumption while meeting the performance requirements of dynamic applications. By changing the frequency and voltage of the CPU dynamically, the performance can be adjusted to the current workload. Dynamic Frequency Scaling (DFS) and Dynamic Voltage Scaling (DVS) are variants of this.

To determine a suitable scaling factor, it is important to know how much power the system uses while performing a given task at different operating points, as well as the time the system is allowed to spend doing it. Information like task arrival time, deadline, and workload, are usually not known in advance with general-purpose computing systems, but with embedded systems they might be, which makes a good foundation for system scenario-based design. Traditional DVFS systems tend to work either fully dynamically, only based on run-time information [22, 23] or statically based on design-time analysis [24]. System scenario-based DVFS is a combination of these approaches. Equation 2.9 in the previous section shows that the energy from dynamic power scales quadratically to the operating voltage, so a lot of energy can be saved by adjusting this factor. Equation 2.9 also shows that dynamic energy is unaffected by reducing the frequency, but if the increased execution time can give less waiting time for the next task, there will be less energy spent while waiting.

The frequency reduction can happen when CPU cycles would otherwise be wasted, e.g., when the CPU is waiting on peripherals. The resulting power reduction must be sufficient to both make up for the time it takes to switch and for the extra time it takes to run the task at a lower frequency. Without being able to reduce voltage, lowering the frequency is not likely to lead to a reduction in energy. It can be seen from Equation 2.9 that when assuming no static power, the energy consumption is not dependent on time or frequency, so with constant voltage the energy is actually constant for a given task, no matter how fast it is performed. The only difference between performing a task fast and slow is thus the energy consumed while waiting for the next task (idle time). The energy contribution from static power is independent of frequency [20], but increases linearly with the execution time (Equation 2.7). However, when voltage is also reduced, a significant energy impact can be obtained due to the V^2 factor in the dynamic power equation. Exactly how much a voltage reduction impacts the total power consumption of

a platform varies from system to system. There are many publications on this, but most are using quite platform-specific models [10, 11, 19].

Snowdon et al. describes in [10] a model that can predict the execution time and power consumption at any voltage-frequency operating point, given the workload execution at another voltage-frequency operating point, on a low-power embedded platform. The model is based on using the platform's performance counters, which monitor and count certain events at run-time, e.g., bus transactions and CPU stall cycles. It is concluded that the limited number of performance counters cannot capture the whole system behavior. Proper model calibration is therefore essential for accurate results.

Castagnetti et al. [19] show that energy can be saved using DVFS, but that the gains are much smaller than the simplistic models predict, as there is a portion of the power consumption of the chip that does not scale with the frequency. Reducing the processor frequency leads to reduced energy dissipation in the core, but the increased execution time can also result in increased energy consumption from other components. Miyoshi et al. [25] analyzes the run-time effects of frequency scaling on performance, power and energy. Their results show that when voltage is kept constant, on a PowerPC-based embedded system it is more energy efficient to run at a low frequency, while on a Pentium-based high performance PC system, it is more energy efficient to run at the highest performance state (Race to Idle). This is explained by introducing a metric called *Critical power slope*. This metric determines whether it is beneficial to run fast and wait in idle-mode, or to run slow and thus minimize idle-mode time. Voltage scaling is also discussed to some extent.

2.4 Memory organizations

Memory organization has been recognized as an important design topic when it comes to reducing energy consumption. In general, memory contributes considerably to the total system energy consumption; from 35% to 65% for different architectures according to [13]. This section briefly describes some of the techniques that are used to increase the performance and reduce the power consumption of memories today.

The fast advances within processor performance during the last few decades has lead to a big gap between processor and memory bandwidth [26]. The introduction of *memory hierarchies* has become the major resolution to this problem. A memory hierarchy provides a smart compromise between fast and big memories by organizing memories with different speed and sizes into different levels. Fast memories are expensive, so they are kept small and located close to the CPU. Further away from the CPU, the memories become slower and larger. Smart strategies are used to keep the data that is most likely to be accessed close to the CPU. Usually, the data contained in each level is a subset of the next lower level. The *cache* is at the highest level of the memory hierarchy, i.e. it is the memory closest to the processor, except the processor's internal registers. There can be several cache levels, with increasing access times and sizes as we move down the hierarchy, but all the cache levels are usually integrated on the processor chip. Hardware controlled cache hierarchies are typically incorporated in today's high-end embedded processors [17]. The next level down in the hierarchy is the main memory.

Scratchpad memories can be used as an alternative to cache hierarchies. A scratchpad memory is similar to a level one cache as it is the memory closest to the CPU apart

from the internal registers. As opposed to caches however, there is usually no copy of the scratchpad data in the main memory, so that the memory access latencies are determined by where the data is stored. A system using scratchpads will therefore have non-uniform, but predictable memory access latencies. A cache hierarchy system on the other hand will typically have homogeneous access latency, but with an uncertainty associated to how far down in the hierarchy the data must be searched for. The required logic for a scratchpad memory is significantly less than for a cache of the same size [17]. So far, the scratchpad memories have not made it to mainstream processors, but they can be found in embedded systems and special-purpose processors such as GPUs. [26].

A tendency is to design more domain specific and heterogeneous memory architectures, e.g. by making use of scratchpads. With such architectures, the memory access time usually depends on the location of the relevant memory relative to the processor, which is why they are classified as Non-Uniform Memory Access (NUMA) architectures. It is predicted that these designs will become important in the future [27]. Deciding where to map data becomes a more difficult task that has to be handled by the designer or compiler, but much energy can be saved if one is able to exploit the heterogeneity [17, p. 197]. As demonstrated in [12], this can be achieved through the system scenario methodology.

Many modern memory systems support different power modes and power gating, so that memory banks can be either shut off completely, or put in sleep modes on an individual basis. There are SRAM memories that can be put into retention states to save power while not erasing data. Alternatively they can be shut off completely to save more power, but without keeping the data. It is predicted that leakage currents will become a dominant source of energy consumption [4, 17]. By introducing memories that support several energy states, e.g., standby, sleep and power down, at least the memory leakage power can be reduced drastically. A memory architecture is described in [17, p.197-200] where the memory is partitioned into memory sub-arrays, with energy states that are managed by a hardware controller. It takes ten cycles to wake up a memory sub-array from a sleep state, in which the energy consumption is 162 times less than in standby state. An overview of the current state of reconfigurable memory can be found in [28]. It has been shown that accessing data in smaller memories requires less energy than big memories, which is a good argument for partitioning memory into several memory banks. Additionally, this enables a parallelization of memory accesses, and more fine-grained control of memory sleep states.

The memory use characteristics of an application can have big variations, e.g., in the number of loads and stores, and the size and use of currently allocated memory. Being able to exploit the dynamic memory requirements of an application can therefore lead to big energy reductions which can have a considerable effect on the total energy consumption. The way data is assigned to available memory banks is an important aspect, affecting both the energy per access, the possibility of data conflicts, and the number of banks that must be active. Techniques for estimating the required memory size to store concurrently alive elements in an application are presented in [29]. Filippopoulos et al.[13] use detailed memory models to construct a reconfigurable memory platform. Applications are analyzed at design-time to identify different execution paths that cause variations in memory usage. An algorithm for exploring different memory organizations is also presented. This algorithm uses Equation 2.10 to calculate the overall energy consumption of all memory banks in a configuration:

$$
\begin{aligned}
E = \sum_{all}^{memories} & (N_{rd} \cdot E_{Read} \\
& + N_{wr} \cdot E_{Write} \\
& + (T - T_{LightSLeep} - T_{DeepSLeep} - T_{ShutDown}) \cdot P_{leak_{Active}} \\
& + T_{LightSleep} \cdot P_{leak_{LightSleep}} \\
& + T_{DeepSleep} \cdot P_{leak_{DeepSleep}} \\
& + T_{ShutDown} \cdot P_{leak_{ShutDown}} \\
& + N_{SWLight} \cdot E_{LightSleepToActive} \\
& + N_{SWDeep} \cdot E_{DeepSleepToActive} \\
& + N_{SWShutDown} \cdot E_{ShutDownToActive})
\end{aligned} \tag{2.10}
$$

N_{rd} and N_{wr} are the numbers of reads and writes, where the corresponding energy is given by E_{Read} and E_{Write}. T is the total time, and $T_{LightSLeep}$, $T_{DeepSLeep}$ and $T_{ShutDown}$ are the time intervals spent in each of the sleep and shut down modes. Correspondingly, $P_{leak_{Active}}$, $P_{leak_{LightSleep}}$, $P_{leak_{DeepSleep}}$ and $P_{leak_{ShutDown}}$ are the power consumptions of the modes. $N_{SWLight}$, N_{SWDeep}, $N_{SWShutDown}$ are the number of switches to active state, where the energy of each switch are given by $E_{LightSleepToActive}$, $E_{DeepSleepToActive}$ and $E_{ShutDownToActive}$. Note that the memory energy consumption in Equation 2.10 depends on the execution time, so that introducing DVFS into the scenario design will affect the memory energy.

2.5 SPEC CPU 2006

SPEC CPU™ 2006 [15] is a benchmark suite developed from real-life applications and workloads. It is intended as a measure for comparing computational intensive performance across a wide range of hardware. A variety of different benchmarks are included for measuring the performance of a system's processor, memory subsystem and compiler. Composing a representative workload was important to the designers of the SPEC benchmark [15]; the workload has to be both diverse enough to capture relevant run cases, and at the same time relatively compact as even high-level architectural simulations are time consuming. The size, both in terms of bytes of code and number of instructions varies a lot (some of the benchmarks are in the range of a few trillion instructions[30]). The benchmark suite can be divided into integer benchmarks (SPECint®2006) and floating point benchmarks (SPECfp®2006), consisting of 12 and 19 benchmark tests respectively. In [31], all the benchmarks are characterized and analyzed in terms of instruction composition and performance, viewing the workload from each benchmark as a whole.

2.6 Other related work

As mentioned, Borkar et al. [4] predicts a trend towards heterogeneous microprocessors. Transistor size can still be decreased, but the transistor threshold voltage and thereby also

the transistor power can not. To still be able to improve power and performance characteristics, a solution is to include processing elements with different power and performance characteristics on a chip. Then the most optimal processing element can be chosen to execute a task or application, while the unused processing elements are powered off. The cost of moving data will have a critical effect on the achievable performance of such a system. Data must be moved efficiently up and down the memory hierarchy, as well as between processing elements that are used. When the supply voltage is downscaled, and thereby also the frequency, the energy spent on moving data becomes more dominant. This effect can be reduced by keeping data as local as possible to the processor, e.g. by increasing the register and cache sizes. This is contrary to the general idea of keeping registers small and fast, but it makes sense with a downscaled CPU frequency[4]. Borkar et al. also recommends DVFS for smaller cores used for throughput, but not large cores as it would *"dramatically reduce single-thread performance"*.

The activity patterns of servers seem promising for system scenario design. According to Barroso et al. [32] servers usual operate at 10-50% of maximum utilization. Being completely idle or operating near the level of maximum utilization is very rare. The energy efficiency at the most common operating point (20-30% utilization) is actually less than half the energy efficiency at peak performance, which indicates a mismatch that must be addressed. The CPU fraction of total server power is however no longer dominant, so it is important that a power down-scale can happen also at the other components, e.g., network switches, disk drives and DRAMs. Disk drives are unfortunately infeasible to spin-down due to the latency and energy penalties of spinning them up again [32].

DVFS is a well-documented power saving technique, but run-time dynamic application is limited, and the development with complex platforms, transistor scaling and improved idle/sleep modes reduces its relevancy. In [9] it is even predicted that manufacturers will abandon DVFS in favour of ultra low-power sleep modes. However, by applying the DVFS technique within a system scenario environment, the shortcomings can be alleviated.

Many applications have resource requirements that are more or less platform-dependent due to different instruction sets, functional units and more. To avoid getting tied to one specific platform, Hamers et al. [33] suggests two different ways of characterizing an application: Using domain knowledge or automated characterization. Using domain knowledge means to use ones understanding of an application to identify what causes its dynamism. E.g. for a H.264 video stream decoder, the size of a frame causes differences in the workload independently of the platform, and is therefore used as a relative measure on performance requirement. Automated characterization requires only limited domain knowledge. A functional execution profile of each frame is collected using an instrumented platform that records the control-flow behavior at run-time. This characterization is data-dependent, and not platform-dependent [33].

Chapter 3

Platform and profilers

Selecting a suitable platform is crucial in the system scenario design methodology. The platform should be highly configurable, even at run-time, while being able to meet the demands of the application. In this work, a platform with only one computational unit is considered. Having several independent computational units allows for much more flexibility; tasks might be scheduled to different cores depending on required performance, and cores can be turned on and off depending on current (and future) application requirements.

3.1 Intel Pentium M

An Intel Pentium M processor is used as reference processor in this work. Pentium M is a family of 32-bit x86 single-core microprocessors that are well suited for DVFS [34]. The current and future generations of Intel Pentium M Processors support the *SpeedStep Technology*. This technology allows the application software to control the operating frequency and voltage of the processor in order to adapt the CPU performance to changing workload and requirements.

Table 3.1 shows the available frequency and voltage combinations of Intel Pentium M. These are from now on referred to as *operating points*. The processor core unavailability time is 10µs for a change of operating point [34].

Table 3.1: Intel SpeedStep operating points as provided by [34]. TDP is explained in Subsection 3.1.1

Frequency [GHz]	**Voltage [V]**	**TDP [W]**
1.6	1.484	24.5
1.4	1.420	
1.2	1.276	
1.0	1.164	
0.8	1.036	
0.6	0.956	6

3.1.1 Power estimations

The only power characteristics found for Intel Pentium M is the Thermal Design Power (TDP). TDP is defined as the peak thermal power, and is typically used when designing CPU cooling systems. The TDP is not necessarily the same as the maximum power; it is possible for the processor to consume more than the TDP for a time interval short enough to make it "thermally insignificant"[35]. In [36], the TDP of a set of processors is compared to the geometric mean power consumed with a suite of server workloads. In average, the TDP is 30% higher than the mean power consumption. The relative CPU power reductions presented in this work are calculated based on the assumption that the average power consumption will be proportional to the TDP at each operating point. Unfortunately, Intel only gives the TDP of 1.6GHz and 0.6GHz operation. According to [34], the dissipated power will be approximately proportional to the square of the voltage between these operating points. This statement could not be verified for the given TDP values, probably because it is based on a simplified power equation that does not account for static power (Equation 2.5). The remaining TDP values are therefore calculated more accurately in the rest of this subsection. The following assumptions are made:

- $TDP = AC \cdot f \cdot V^2 + P_{static}$ (Equation 2.3 and 2.5)
- $AC = k_1$ is constant
- P_{static} is proportional to V^2 (very simplified; see Section 2.2.1), i.e. $P_{static} = k_2 \cdot V^2$ where k_2 is a constant

This gives:

$$\begin{aligned} TDP &= AC \cdot f \cdot V^2 + P_{static} && (3.1) \\ &= k_1 \cdot f \cdot V^2 + k_2 \cdot V^2 && (3.2) \end{aligned}$$

The operating voltages increases almost linearly with increased frequencies, so the relationship between voltage and frequency is simplified through linear regression to the expression shown in Equation 3.3 and Figure 3.1. All calculations are made with frequency in GHz for better readability.

$$f = 1.779 \cdot V - 1.075 \qquad [\text{GHz}] \qquad (3.3)$$

By inserting Equation 3.3 into 3.2 the following expression is found:

$$\begin{aligned} TDP &= k_1 \cdot (1.779 \cdot V - 1.075) \cdot V^2 + k_2 \cdot V^2 && (3.4) \\ &= k_1 \cdot 1.779 \cdot V^3 + (k_2 - k_1 \cdot 1.075) \cdot V^2 && (3.5) \\ &= k_3 \cdot V^3 + k_4 \cdot V^2 && (3.6) \end{aligned}$$

Then, by using these two conditions from Table 3.1;

$$\begin{aligned} TDP &= 24.5 \text{ when } V = 1.484 \\ TDP &= 6 \quad\; \text{ when } V = 0.956 \end{aligned}$$

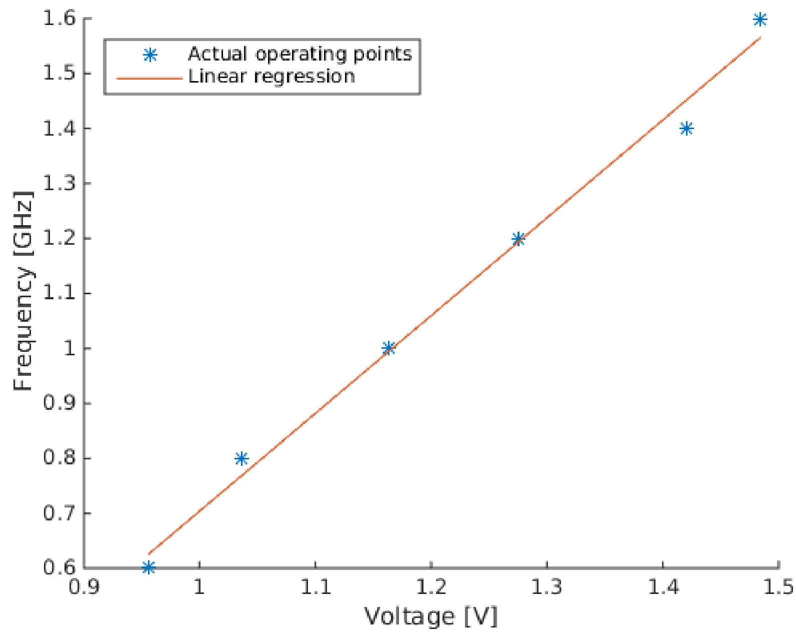


Figure 3.1: Linear regression of Intel Pentium M operating points.

k_3 and k_4 are found:

$$k_3 = 8.636 \tag{3.7}$$

$$k_4 = -1.691 \tag{3.8}$$

This gives:

$$TDP = 8.636 \cdot V^3 - 1.691 \cdot V^2 \tag{3.9}$$

Alternatively, if assuming that the static power consumption can be neglected, the power consumption at each operating point can be calculated using Equation 2.5 ($P_{dynamic} = AC \cdot f \cdot V^2$). The AC product found from the TDP calculations is used ($k_1 = 4.85$), as well as the voltage-frequency pairs given in Table 3.1. Figure 3.2 compares this $P_{dynamic}$ to the estimated TDP of Intel Pentium M.

Table 3.2: Intel SpeedStep operating points with the estimated TDP.

Frequency [GHz]	**Voltage [V]**	**TDP [W]**
1.6	1.484	24.5
1.4	1.420	21.3
1.2	1.276	15.2
1.0	1.164	11.3
0.8	1.036	7.8
0.6	0.956	6.0

Given perfect conditions (e.g. no waiting for memory or synchronization) the CPU's energy consumption when running an application can be reduced significantly by using

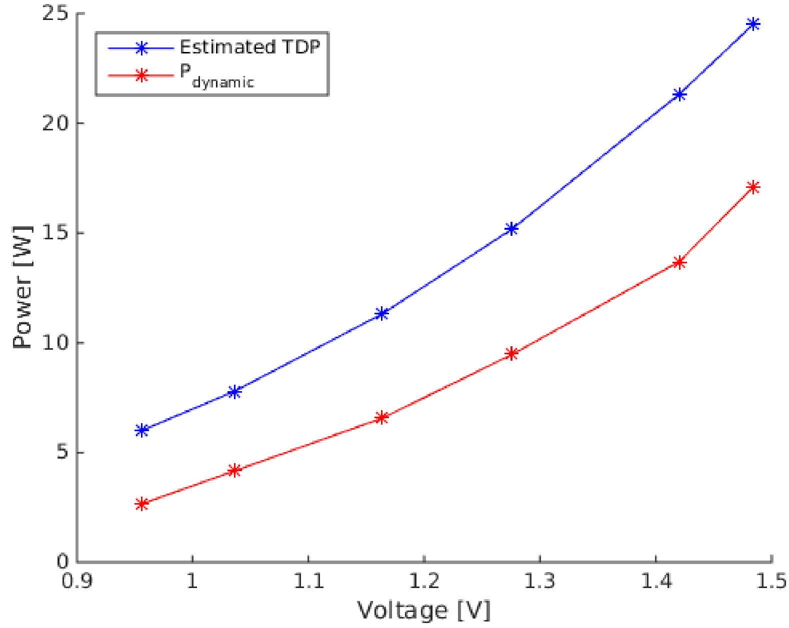


Figure 3.2: Estimated TDP of the Intel Pentium M operating points compared to calculated $P_{dynamic}$

.

the low-power operating points. As shown in Equation 3.13, if the lowest performance operating point is used instead of the highest, the CPU energy consumption is reduced with 34.7%. If the increased execution time means reduced idle time (e.g., waiting for synchronization or peripherals), the energy reductions will be even bigger.

$$E = P \cdot T = P \cdot \frac{cycles}{frequency} \tag{3.10}$$

$$E_{highPerformance} = 24.5 \cdot \frac{cycles}{1.6 \cdot 10^9} = cycles \cdot 15.3125 \cdot 10^{-9} \tag{3.11}$$

$$E_{lowPerformance} = 6 \cdot \frac{cycles}{0.6 \cdot 10^9} = cycles \cdot 10 \cdot 10^{-9} \tag{3.12}$$

$$(1 - \frac{E_{lowPerformance}}{E_{highPerformance}}) \cdot 100\% = 34.7\% \tag{3.13}$$

It is here assumed that the number of CPU cycles ($cycles$) is constant, regardless of the operating point.

The operating points of the Intel Pentium M can also be presented as in Figure 3.3, by comparing the CPU energy per clock cycle of each operating point. Here it can be seen that, when trading off time against energy, all TDP operating points except 0.6GHz (to the right) are Pareto-optimal. The energy reduction found in Equation 3.13 is therefore even higher (36.4%) if using 0.8GHz operation as the *lowPerformance* mode. Reduced CPU idle time can still make it beneficial to operate at 0.6GHz. All the $P_{dynamic}$ operating points are Pareto-optimal, which is as expected considering Equation 2.5. Many assumptions have been made in order to estimate the missing TDPs, so the actual TDPs might not

correspond entirely with the estimated. It is for example possible that the estimate of the 0.6GHz operating point is not Pareto-optimal because the estimate of the 0.8GHz operating point is more energy efficient than it should be. However, accurate CPU energy estimation is not part of the scope of this thesis, so further investigation of this is left for future work. The TDP operating points will be used to estimate the system scenario results.

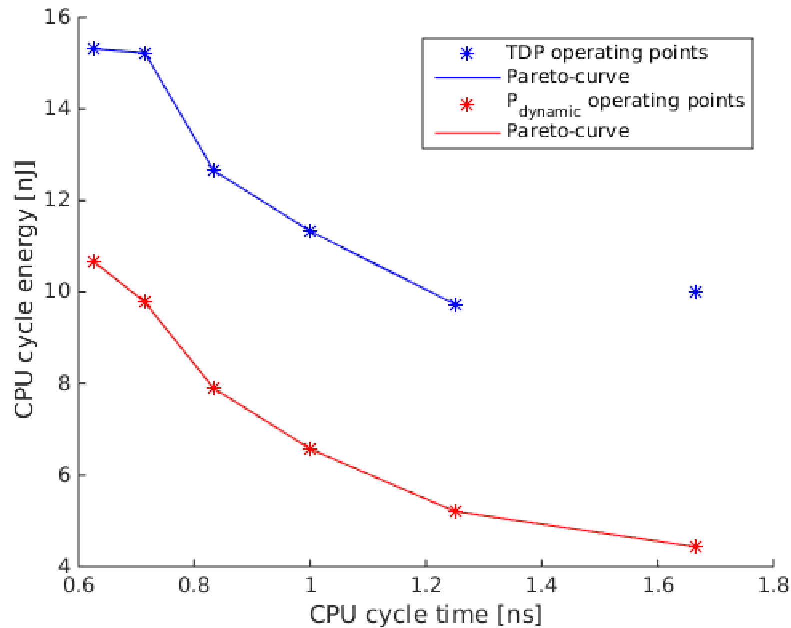


Figure 3.3: CPU cycle energy as a function of cycle time, for all the TDP and $P_{dynamic}$ operating points. The line is drawn between the Pareto-optimal operating points.

3.2 Memory Platform

The memory platform used in this work is a theoretically constructed platform with several configuration possibilities. As in [13], the memory platform can be configured at design-time to include a variety of memory banks of different sizes, depending on the application requirements. At run-time, these memory banks can be shut down or put into various sleep modes in order to save power. In other words, there is reconfigurability both at design-time and run-time to be explored. In this work only two memory bank states are considered; active state and shut-down state. This is done because it suits the considered applications, and because it simplifies the computations somewhat. The memory characteristics used for estimating the memory power consumption are partly taken from CACTI 5.3 models obtained using the CACTI web interface [37, 38], and the memory bank characteristics in [13].

Originally, the considered sizes were 32MB, 16MB, 8MB, 4MB, 2MB, 1MB and 512KB, modelled with the default CACTI cache model. However, to be able to test the scenario methodology on benchmarks with bigger memory requirements, some RAM memory banks are added. The memory requirements of the considered applications vary from

10MB up to 390MB. This much memory implemented as cache is not considered realistic, which motivates the need for (at least) two different kinds of memory banks. The default CACTI cache model is used for memory sizes up to 32MB, and a CACTI RAM model for bigger memory sizes. The read energy and stand-by power for each memory bank are obtained from these models. The exact input parameters to CACTI are listed in Appendix A.6. The CACTI web interface does not provide shut-down power, so the relationship between standby power and shut-down power in [13] is used to obtain shut-down power for the CACTI memories. The final memory bank characteristics are shown in Table 3.2.

Table 3.3: Memory characteristics of relevant memory banks.

Memory size	P_{active} [W]	$P_{shutDown}$ [W]	E_{read} [nJ]	Memory type
512MB	17.231	7.183	7.715	RAM
256MB	8.606	3.587	4.631	RAM
128MB	4.380	1.826	2.864	RAM
64MB	2.184	0.910	1.732	RAM
32MB	14.259	5.944	2.330	Cache
16MB	7.889	3.289	2.498	Cache
8MB	3.905	1.628	2.188	Cache
4MB	1.905	0.794	0.815	Cache
2MB	0.940	0.392	0.652	Cache
1MB	0.485	0.202	0.366	Cache
512kB	0.272	0.113	0.236	Cache

The CACTI web interface does not give energy per memory write. It is assumed that this energy is equal to the energy per read, which should be a fair assumption according to the memory characteristics given in [13]. All of the memory banks have different access times, ranging from 0.973ns to 29.996ns. Proper pipelining of the memory accesses *might* minimize the effects of this. This is however a challenge that is not addressed in this work and should be investigated in future work. As mentioned in Section 2.4 there is a tendency to design more domain specific and heterogeneous memory architectures where the memory access time depends on the location of the relevant memory relative to the processor (NUMA). These concepts might be helpful in a memory system scenario context.

The number of extra wires and combinational logic from partitioning the memory into several banks will use extra resources such as energy and area. As in [13], it is however assumed that the increased interconnect overhead from this can be neglected when considering architectures with five banks or less.

3.3 Scenario Manager

The *scenario manager* comprises the run-time scenario control system, i.e. the scenario prediction and switching mechanisms. This section briefly describes the suggested software and hardware implementations for the system scenario designs that will be presented in Chapter 5. The software alternative is chosen due to the time limitations of this work,

but as we shall see the hardware alternative might be preferable in terms of energy efficiency and reliability.

3.3.1 Software implementation

When implemented in software, the scenario lookup table is stored in general purpose memory. How much space it requires depends on the number of entries in the table, i.e., the number of scenarios, and how the scenarios are recognized. As an example, consider a lookup table consisting of five scenarios, where each scenario is mapped to two RTS parameter values. This can be implemented as a two-dimensional array, as shown in Table 3.4. Provided that each integer is represented by 4 bytes, a 3x5 integer array in C will require only $3 \cdot 5 \cdot 4\text{B} = 60\text{B}$ of memory.

Table 3.4: Example of scenario lookup table with two RTS parameters.

RTS parameter a	**RTS parameter b**	**Scenario**
1	1	1
1	2	2
2	1	3
2	2	4
-	-	Backup

The scenario control mechanism is implemented in functions, typically invoked at specific locations in the source code, triggered by external events, or by a periodic interrupt. In this work, the system scenario framework developed by Yassin et al. [39] is used. The scenario prediction is implemented in the *AMU function*, and the scenario switching in the *PAM function*. When the RTS parameters that are used to predict a scenario change are updated with a new value, the AMU function checks these against a lookup table to see if the current scenario will continue to be optimal or not. If not, and if the costs of switching to a better suited scenario is outweighed by the gains, the PAM function is triggered and executes the switch.

3.3.2 Hardware implementation

A disadvantage with the AMU and PAM functions in a single-core system is that the application is stalled when these are executed [39]. The scenario management can instead be performed by dedicated logic, rather than through a set of instructions that steals CPU time. With the software implementations used in the current work, the CPU will spend more time doing the same amount of overhead when in a low performance scenario than it will in a high performance scenario. A hardware implementation of the prediction and switching mechanisms can give more deterministic and reliable execution. It can also allow for more or less constantly monitoring the RTS parameters, which is useful in applications where the RTS can change at any time. Furthermore, the energy overhead can be minimized by using dedicated logic. This is somewhat elaborated in [11]. A state machine, as seen in Figure 3.4, and a lookup table with the RTS to scenario mapping, as in Figure 3.5, are therefore suggested for implementation in future work.

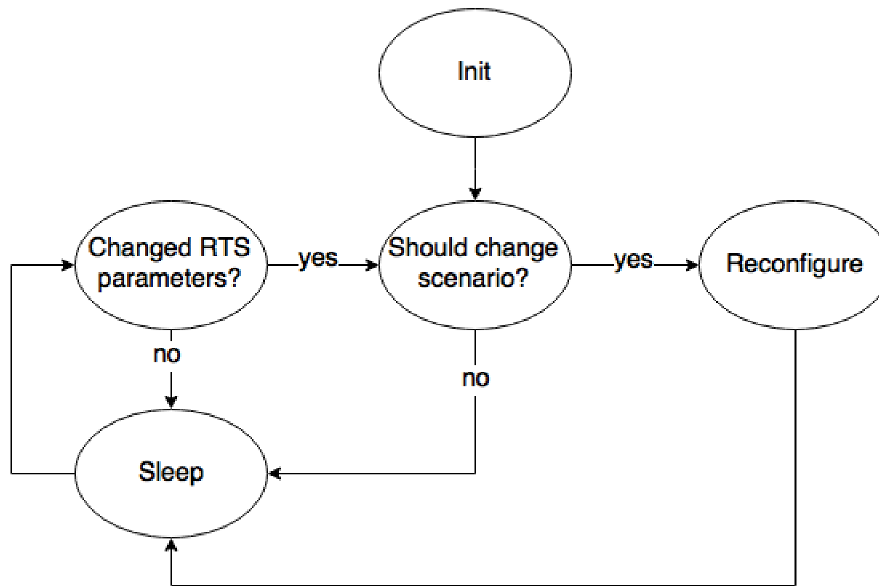


Figure 3.4: Scenario control state machine.

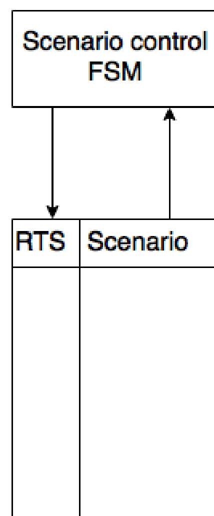


Figure 3.5: Scenario control and the lookup table.

It should be noted that adding logic to the system will increase the total energy consumption somewhat. This additional cost should be compared against the cost of the software alternative, and weighted against the improved prediction performance that the hardware alternative can give.

3.4 Profilers

The profiling results in Chapter 5 are obtained using Zoom, gprof, Cachegrind and Massif. This section gives an overview of these profilers and how they were used.

3.4.1 Zoom

Zoom is a system-wide profiler that samples call stacks, scheduler traces, and several kinds of CPU events, such as CPU cycles, instructions, branches and cache misses. The profiling overhead is very low and no source code modifications are needed [40]. Zoom can profile the entire system with all executing processes and threads, or only the threads within a single process. The thread of interest can then easily be extracted from the rest of the profile. Additionally, Zoom offers a code browser that can show where in the code the samples were taken, at source-code and assembly level. The profiler can be controlled programmatically through the ZoomScript API, so that the profiler can be started and stopped at specific locations in the source code [41].

In this work, Zoom is used to find the number of CPU cycles necessary to execute whole applications, and smaller parts of applications. It is also used for tracing function calls and browsing assembly samples.

Accuracy

The SPEC CPU 2006 benchmark HMMER with input retro.hmm and swiss41 (see Section 5.2) was run 30 times and the total number of CPU cycles was recorded with Zoom. The maximum deviation from the mean was 3%.

3.4.2 Valgrind Massif

Massif [42] is a tool which is part of the Valgrind instrumentation framework. Massif profiles the heap memory that an application uses, both the actually utilized space and the extra bytes that are used for purposes like alignment and book-keeping. The information can be visualized as a graph by using the `ms_print` utility, which can show allocated memory as a function of time, instructions or bytes. Massif also records which parts of the application that are responsible for the allocation. When profiling with Massif, the execution time becomes about 20 times slower than normal.

Massif is used to find the memory size requirements of the applications, and for locating where in the source code the allocations take place.

3.4.3 Valgrind Cachegrind

Cachegrind is a cache and branch-prediction profiler that simulates how an application interacts with the cache hierarchy and branch predictor [43]. It also gathers loads and stores statistics at source code and assembly-level. When profiling with Cachegrind, the execution time becomes between 20 and 100 times slower than normal.

In this work, Cachegrind is only used to find the number of loads and stores. Intel's VTune was used initially for this purpose, but when the free trial period expired, Cachegrind was used instead. The obtained number of loads and stores were the same.

Chapter 4

Applied methodology

This chapter presents a modification of the traditional system scenario methodology described in Chapter 2. The methodology is the generalized result of trying to identify and exploit the dynamism in CPU and memory-intensive applications, without ending up with sub-optimal results. The applications and the system scenario design results are presented in Chapter 5.

The platform described in Chapter 3 is assumed for the system scenario development. The suggested memory platform provides a set of memory banks that can be included in the implementation if feasible. At design-time, the memory requirements of the application can thus be used to tailor the memory platform to the application. For the selected memory configuration, the number of active memory banks and shut-down memory banks is managed at run-time, i.e this is the memory system knob. The suggested CPU has a set of operating points with different power and performance characteristics, where the active operating point is managed at run-time. The CPU system knob is therefore the operating point of the CPU. To control these two system knobs at run-time without too much overhead, the optimal combined settings must first be found at design-time.

Figure 4.1 shows design-time workflow that is developed and followed during this work. To alleviate the complexity from taking several costs into account, the methodology suggested here is to first consider each cost separately. I.e. the identification and prediction steps of the system scenario methodology are carried out with each system knob individually, and the identified RTSs are grouped into sub-scenarios specific to each system-knob. The methodology is developed and explained with DVFS and dynamic memory reconfigurations in mind, but it can also be used for other system knobs as the main concepts are generic.

Note that limiting the number of sub-scenarios is important. For example, if only three memory sub-scenarios and three DVFS sub-scenarios are identified, we will already have up to nine possible combinations of these. The next step is the scenario exploration, where all the scenario configurations are explored to find the best suited platform configuration for each sub-scenario. This step is not a part of the original methodology. As explained in [6], separate optimizations of each scenario can lead to separate systems that give too much switching overhead in the final system. With the assumed platform however, all system knob settings can be combined without much overhead. This might not be possible with other platforms, indicating a weakness with the applied methodology. The scenario exploitation, which originally follows *after* the prediction step, is in this work performed

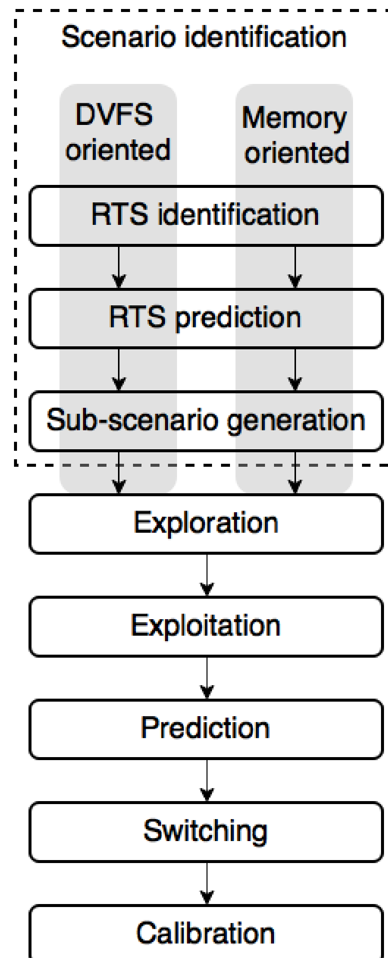


Figure 4.1: Applied workflow.

before the prediction. The reason is that many of the scenarios will typically be merged during the exploitations, so to create prediction mechanisms before this can lead to much unnecessary work.

The usefulness of the suggested modifications might be limited to applications and platforms that are similar to the ones that are considered. The original methodology is more general, and could be better suited if other system knobs are considered. It can for example be a waste of time to first consider each system knob separately (as in "DVFS oriented" and "Memory oriented" in Figure 4.1) if the setting of one system knob immediately excludes several settings of another system knob.

4.1 System requirements and characteristics

Before starting the system scenario development, one must be familiar with the application and the platform in order to know what kind of dynamism to look for and how it can be exploited. It is important to determine the performance requirements of the system; e.g., whether the quality and accuracy of the result are most important, or if it can be traded off against energy efficiency. Preferences like fast computation, high throughput and responsiveness should also be considered. A prioritized list can be useful if there are no strict requirements.

The energy consumption of the platform to be used should be properly characterized for the whole range of relevant configurations (briefly demonstrated in Chapter 3). The power consumption and performance must be known for the relevant execution and sleep modes when various operating voltages and frequencies are used, and possibly with different modules turned on and off. The costs of switching between the different settings should also be known, at least for the worst case. Sub-optimal configurations, and configuration combinations that are too complicated for run-time switching can be omitted. For example, it is usually optimal to run at the highest frequency available at each operating voltage level in terms of energy efficiency. Different memory configurations should be characterized when the application can benefit from this (i.e. has dynamic memory requirements that affect the total energy considerably). The expected delay of waking up a memory bank should then be known, as well as the energy for loads, stores and the power consumed in each relevant mode for all memory bank sizes.

Regarding the application, it can be difficult, or even impossible to find the whole set of inputs and situations that might occur in real-life use of the final system. A representative set for the profiling should at least contain the most frequent behavior, and preferably some corner cases. As opposed to RTS identification from static analysis, the profiling-based scenario identification is not complete unless all possible RTSs are tested. If an unfamiliar RTS happens at run-time however, a back-up scenario can be used, as was explained in Section 2.1.1.

4.2 Scenario identification

As described in Section 2.1, the first step in scenario development is to identify the scenarios in the application. The scenario identification step can be divided into several smaller steps depending on the type of dynamism that is searched for. Gheorghita et

al. [6] divide the identification step into RTS parameter discovery and RTS clustering. The current work suggests to additionally separate the scenario identification by the type of dynamism that is searched for. The RTS clustering is also performed somewhat differently. This section describes the suggested approach, which consists of one procedure for DVFS dynamism and one for memory dynamism. Some modifications, e.g. other profiling techniques, will be needed to use the procedures with other kinds of dynamism. The utilized profilers are described in Section 3.4. The procedures can be performed in parallel and are platform independent.

It has been attempted to make the procedures somewhat application-independent, but the diversity of dynamic applications makes it very difficult to create a procedure that is general and still useful.

4.2.1 DVFS dynamism

RTS identification and prediction

The procedure listed below is the result of characterizing the CPU dynamism in an application through gray-box modelling (see Section 2.1.5). By partitioning the execution of the application and separating the dynamic and static parts, the idea is that it will be easier to more precisely determine the sources of dynamism.

- Profile the application with a representative subset of the possible inputs, using a profiler such as GNU's gprof [44] or RotateRight Zoom [40]. In the case of DVFS, all situations requiring different number of clock cycles are separate RTSs. Try to find patterns between CPU cycles (or execution time), input data and the different RTSs. If there are many possible RTSs, an expression relating the RTS parameter to the resource usage is preferable. This can be found by doing a regression analysis with the RTS parameter and the measured resource usage (CPU cycles).

- Study the source code and the different sets of inputs and situations to determine the variables that lead to dynamism. There must be some way of predicting the dynamism in order to take proper advantage of it. For straightforward applications (like the prime number checker described in Section 5.1), it can be sufficient to perform these two steps, but often it is the case that no satisfying correlation is found from this.

- Partition the application. By dividing the application's execution into several smaller parts, these can be analyzed separately and it can be easier to discover the sources of dynamism. A natural way of dividing an application is by its function calls, starting on top of the function call hierarchy:

  - From the initial profiling, identify the functions that demand the most of the resources. Only the functions that are responsible for considerable amounts of the resource usage needs further investigation. This usually rules out much of the code.

  - Functions requiring the same amount of resources regardless of input or situation can also be left out from further analysis as these can be represented as black boxes.

- Profile the remaining functions separately (e.g. in Zoom). A function called from different places in the code should be profiled separately once for each call, as it might behave differently depending on the state of the application. Profiling here is the same as in the first step. As before, if any of the functions have a CPU cycle count independent of RTS, they can be left out from further analysis as black-boxes. Check for correlation between input and the number of times a function is called, and the resource use of the function. Often it is the case that resource use depends on parameters found during execution. Study the code to identify this.

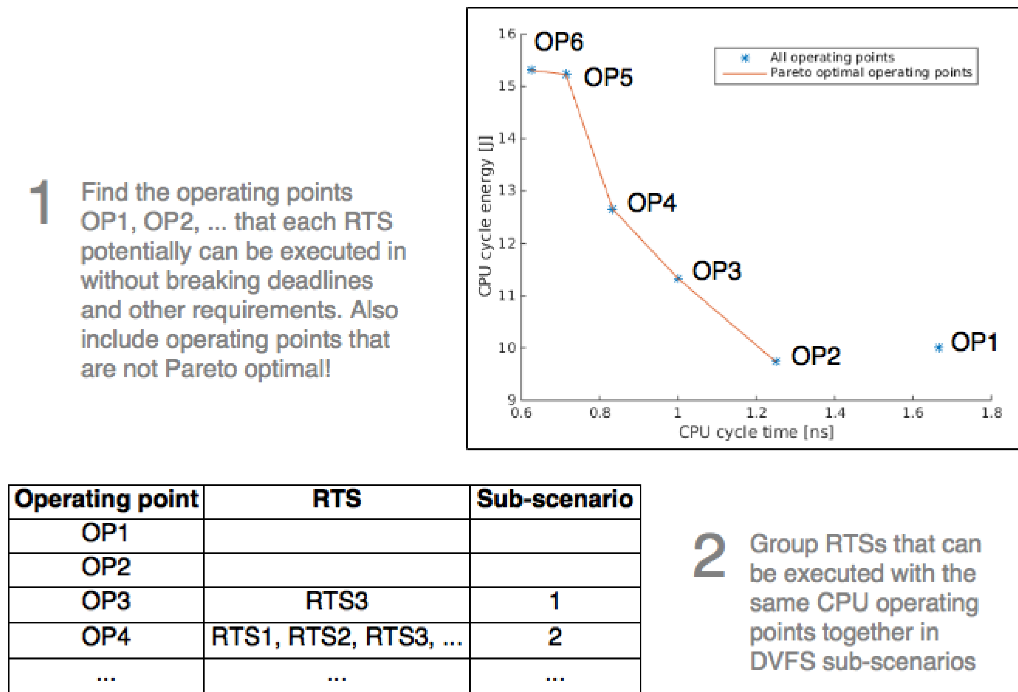
- Further divide the function calls if the results were not satisfying. For sub-function partitioning, look for control structures like for and while loops. Again, check for correlation between input and the number of times a loop is executed, and the resource use of the loop. Often the operations performed within loops can be modelled as black-boxes. Some profilers, like gprof and Zoom offer source code line annotation of CPU cycles, which can be useful here. From this, an expression for the required CPU cycles given some input parameters can be found.

  - As an example, consider a loop where the ending condition depends on a parameter of the current input. Find the number of CPU cycles that this loop is responsible for (from the mentioned CPU cycle line annotation) for different inputs. Also find the total number of iterations for the different inputs, e.g. by modifying the source code to print this. Then, by dividing the cycle count on the number of iterations for each input, the average cycle count for one iteration can be found. If this is consistent, the loop body can be considered as a black box with an associated cost. The cycle count of the loop can then be found by multiplying this cost with the predicted number of iterations to find the predicted number of CPU cycles.

- Repeat the partitioning if necessary.

Sub-scenario generation

The coherence between RTS and CPU cycles found in the previous steps are now used to find a set of possible CPU operating points for the identified RTSs (see Figure 4.2, 1). The feasibility of each operating point can depend on which memory configuration is chosen, so even if some of the operating points are not Pareto-optimal, they should still be included. Of course, if it is clear that some operating point cannot be used or if it is infeasible regardless of other system knob settings, it should not be considered further. E.g. in many applications there are deadlines to be kept, which could mean that some operation points cannot be used to execute the most demanding RTSs. To find out this, calculate the maximal number of clock cycles within a deadline at each operating frequency, and compare these to the CPU cycle count of each RTS. For some applications the deadlines might not be fixed, e.g. because they depend on other tasks. The calculations can then possibly be done at run-time instead, but this might increase the overhead significantly. The number of RTSs can make it feasible to group the RTSs somewhat before assigning them to operating points. RTSs with similar resource requirements (CPU cycles) can be grouped together and represented by the most demanding RTS (i.e. worst-case RTS) of that group.



Operating point	RTS	Sub-scenario
OP1		
OP2		
OP3	RTS3	1
OP4	RTS1, RTS2, RTS3, ...	2
...	...	...

Figure 4.2: DVFS sub-scenario generation.

When each RTS has been mapped to all the operating points in which it can be executed, we get the DVFS sub-scenarios (Figure 4.2, 2). If the RTS set is big, the RTSs in each sub-scenario should be represented only by the worst-case RTS of that sub-scenario (if possible). If an expression has been found for the resource requirements given an RTS parameter, the RTS parameter value that characterizes the most demanding RTS of each operating point can be calculated and used to represent that sub-scenario. When the RTS set is limited, we can treat each RTS individually. The resulting table will be used at design-time during the scenario exploration and exploitation.

4.2.2 Memory dynamism

A platform like the one described in Section 3.2 is assumed, i.e. where the memory banks that are not in use can be put to sleep or shut off. At design-time we first want to find out which memory banks the platform should consist of; each combination of this is from now on called a *memory configuration*.

RTS identification and prediction

We are looking for two main types of dynamism:

1. Allocation space dynamism: The amount of allocated memory depends on the RTS.
2. Allocation time dynamism: The timing of the memory allocation depends on the RTS.

A combination of the above is also possible. The following procedure is suggested:

- Profile the application with many different inputs using a memory profiler such as Valgrind's Massif. Massif provides a graphical representation of the amount of memory allocations and deallocations performed, and the functions that are responsible for this. If no variations in memory use is shown, there is no memory dynamism to be exploited and the rest of this procedure can be skipped. Otherwise, start identifying the RTSs from the Massif graph. If the amount of allocated memory throughout an application run is fairly constant, the whole execution with a given input can be characterized as one memory RTS. In other cases, if there are big variations in memory allocated, the execution should be split in several smaller memory RTSs. Try to recognize the two kinds of memory dynamism.

- Massif gives the source code locations where the big allocations and deallocations happen. Locate these and find the RTS parameters.

- Find a suitable location/time for the prediction method to take place (the prediction method will be created later). Where are the RTS variables assigned their values? If this does not provide sufficient time to wake up memory before the allocation should happen, or to put memory to sleep after a deallocation, try to rearrange the source code so that the RTS variable assignment happens earlier.

- Find the number of loads and stores for all RTSs. This can be done with a profiler that uses hardware counters, such as Intel VTune. Valgrind's Cachegrind simulator can also be used. Another solution is to extract this information from the source or assembly code. The number of loads and stores will be used to calculate energy consumption for different memory configurations.

- The execution time/number of CPU cycles for all RTSs must also be recorded, as the standby-power of the memory banks depends on this.

Sub-scenario generation

When the RTSs have been identified, the platform must be adapted to the dynamism we want to exploit. This is similar to the DVFS sub-scenario generation described in Subsection 4.2.1. The exact methodology for memory sub-scenarios is however somewhat different, as can be seen in Figure 4.3. The memory platform suggested in Section 3.2 has a number of different memory sizes that can be included or left out from the implementation. The most energy efficient memory configurations (a, b, c in Figure 4.3) can be found through an exhaustive search, e.g. considering all configurations with up to five banks, like in [13]. The only requirement is that the total memory capacity of the memory banks in a configuration is sufficient for the most memory-intensive RTS. The ordering of memory banks has no effect, so the total number of possible combination is given by the binomial formula, $\binom{m}{n}$, where m is the number of memory banks for a configuration and n is the total number of available memory banks. In this work only a few configurations are constructed manually. By studying the memory requirements of the RTSs, only the most promising memory platform configurations are explored instead of considering all possible configurations. The following list gives a few guidelines for designing memory bank configurations:

- Optimize for the common case.

- The total amount of available memory in a configuration should be close to the worst-case memory requirements, as a shut-off memory bank will still consume some power.
- Try to include memory banks that enable little excess active memory for each RTS.
- Keep in mind that it can sometimes be beneficial to use more than one memory bank of each size.

For each memory configuration, create different *active bank combinations* (Figure 4.3, 1) that suits the memory RTSs. An active bank combination is a system knob setting of the memory configuration, i.e. a sub-scenario, to which a number of RTSs can be mapped. Within each memory configuration, each RTS is mapped to the best suited active bank combination (Figure 4.3, 2). In this work, this is assumed to be the active bank combination that gives as little excess memory as possible. The RTSs that are mapped to the same active bank combination are now part of the same memory sub-scenario (Figure 4.3, 3). If there are many RTSs, focus on the most memory-intensive RTSs, or group the RTSs based on memory characteristics before mapping them to the active bank combinations. There are different ways of organizing the mapping. One way is to generate a table for each memory sub-scenario that shows how the different memory configurations will be used at run-time. An example of this is shown in Table 4.1. Only two memory bank states (on and off) are shown for simplicity.

Table 4.1: Memory sub-scenario corresponding to RTS2 in Figure 4.3. This table represents the memory sub-scenario for RTSs that require 24MB of active memory. The table shows how many banks of each size that should be on and off in order to get 24MB of active memory, with each of the potential memory configurations. Some of the available banks are not used in the configurations; these are marked gray. Each row in the table gives the possible active bank combination.

Memory bank size	32MB		16MB		8MB		4MB		2MB		1MB	
Memory bank state	On	Off	On	Off	On	Off	On	Off	On	Off	On	Off
Config a	1	0	0	0	0	0	0	0	0	0	0	0
Config b	0	0	2	0	0	0	0	0	0	0	0	0
Config c	0	0	1	0	1	3						
...												

4.2.3 Scenario identification summary

Table 4.2 summarizes the characteristics that should be found during scenario identification. The next section describes how the optimal combination of all the system settings are found from these characteristics.

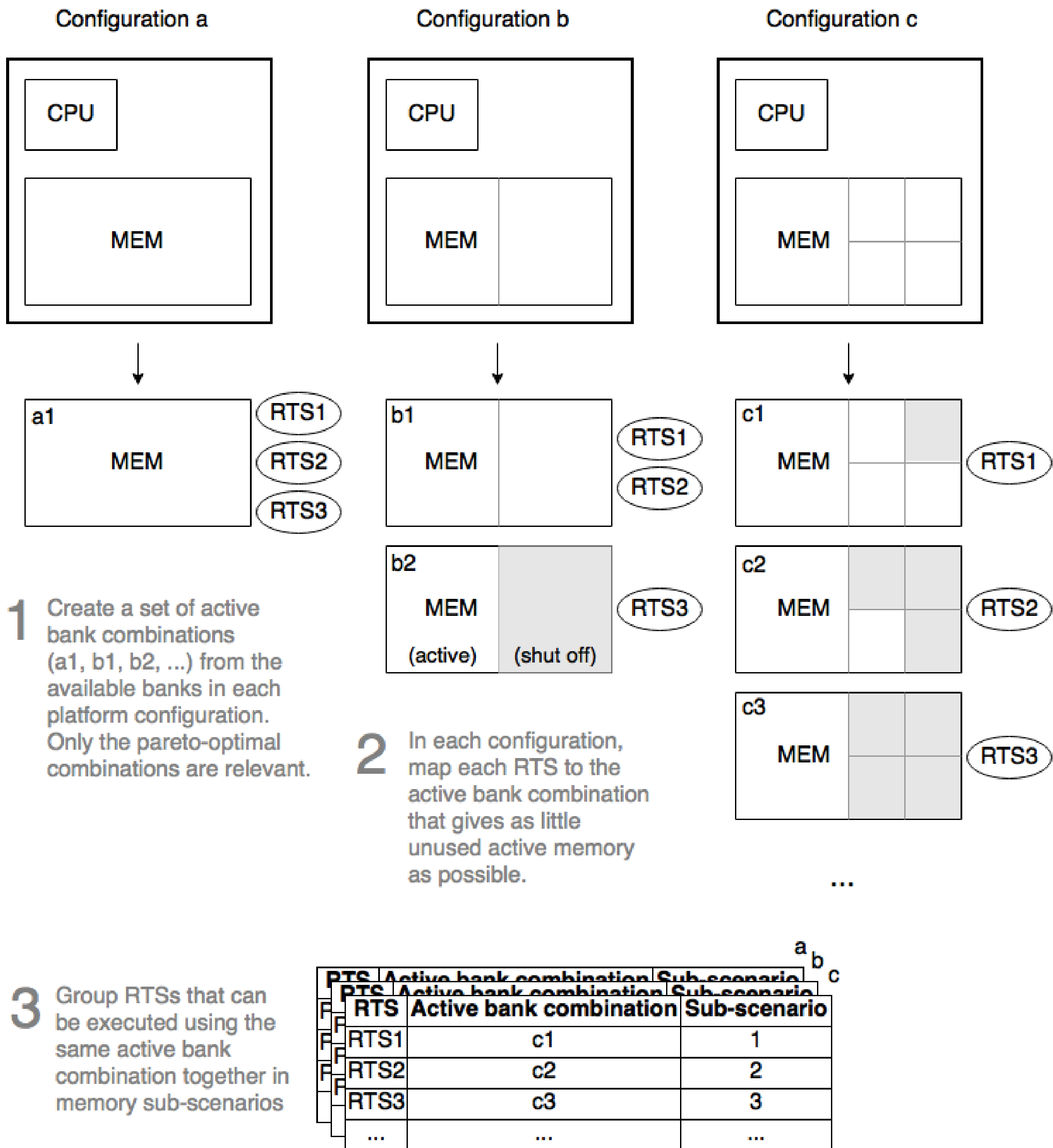


Figure 4.3: Memory sub-scenario generation.

Table 4.2: RTS characteristics, either from profiling all RTSs, identifying an expression or generalization that can be used to estimate the RTS characteristic, or from a combination.

RTS characteristic	How to find it
RTS parameter value	Code inspection
Maximum allocated memory	Valgrind Massif
Number of loads and stores	Intel VTune, Valgrind Cachegrind
Execution time/CPU cycles	Gprof, Zoom

4.3 Exploration and exploitation

Now that the CPU and memory dynamism of the application has been characterized separately, it is time to explore the different sub-scenario combinations. To find the most feasible platform configurations for an RTS or a group of RTSs (sub-scenario) it is important to consider the total cost with all the dynamism to be exploited. As an example, consider an RTS that takes little CPU time. If this RTS is executed in a high-performance operating point, the energy saved by shutting off unused memory banks during this RTS will be very little. It might however be preferable to run the RTS at a lower frequency to reduce CPU power, and then the impact of shutting off unused memory can become considerable.

Algorithm 1 and 2 are developed to solve this two-dimensional optimization problem. First, Algorithm 1 is used to explore the different system settings for each RTS separately. Second, Algorithm 2 performs the exploitation by finding the the optimal system settings considering a representative set of RTSs. Since optimizing for only one kind of dynamism at a time can lead to sub-optimal results, the algorithms estimates and compares the energy consumption of all the possible DVFS and memory configuration combinations. The algorithms are illustrated in Figure 4.4, where step 1 and 2 corresponds to Algorithm 1, step 3 corresponds to Algorithm 2, and step 4 shows how the results are used to create the scenario lookup table.

Algorithm 1 The best CPU operating point for all the RTSs with all relevant memory configurations is found. For each memory configuration, the RTSs are then organized according to their optimal operating points in the three dimensional *scenarios* array. The algorithm corresponds to step 1 and 2 in Figure 4.4.

1: **for all** relevant memory configurations *config* **do**
2:     **for all** situations *RTS* **do**
3:         $E_{bestOp} = \infty$
4:         **for all** operation points *op* **do**
5:             $E_{mem} \leftarrow$ *mem_energy(RTS, op, config)*
6:             $E_{CPU} \leftarrow$ *cpu_energy(RTS, op, config)*
7:             $E_{currentOp} \leftarrow E_{mem} + E_{CPU}$
8:             **if** $E_{currentOp} < E_{bestOp}$ **then**
9:                 $E_{bestOp} \leftarrow E_{currentOp}$
10:                 *bestOp* $\leftarrow$ *op*
11:         *scenarios(config, bestOp).append(RTS)*
12:         $E_{bestOp}(config, RTS) \leftarrow E_{bestOp}$

Algorithm 2 This algorithm finds the best memory configuration for a set of RTSs, corresponding to step 3 in Figure 4.4.

1: $E_{bestConfig} = \infty$
2: **for all** relevant memory configurations *config* **do**
3: $\quad E_{currentConfig} \leftarrow 0$
4: $\quad$ **for all** situations RTS in RTS_set **do**
5: $\quad\quad E_{currentConfig} \leftarrow E_{currentConfig} + E_{bestOp}(config, RTS)$
6: $\quad$ **if** $E_{currentConfig} < E_{bestConfig}$ **then**
7: $\quad\quad E_{bestConfig} \leftarrow E_{currentConfig}$
8: $\quad\quad bestConfig \leftarrow config$

For each relevant memory configuration, Algorithm 1 finds the total cost (i.e. energy) of executing each RTS with each of the CPU operating points that can be applied according to the DVFS sub-scenarios found in Section 4.2.1. Each RTS is then mapped to the operation point with the lowest cost (Figure 4.4, 1). Now we have a set of optimal DVFS sub-scenarios for each memory configuration. The total cost of executing each RTS must be saved (Figure 4.4, 2). Algorithm 2 adds together the total cost of executing the RTSs in the RTS_set, with the optimal operation point for each RTS, for each memory configuration. The resulting total costs are compared to find the optimal memory configuration (Figure 4.4, 3). After Algorithm 2, the final step is to combine the DVFS and memory sub-scenarios of the optimal configuration to get the scenario lookup table (Figure 4.4, 4).

Each considered RTS should be input to Algorithm 1. Alternatively only the worst-case RTSs of each sub-scenario can be used. The RTSs are here optimized in isolation, which *can* result in separate optimizations that are too different to be combined in the same system, as already mentioned in the beginning of this chapter. Algorithm 2 is meant to optimize for a *typical* set of RTSs, RTS_set. The ordering of the RTSs in this set does not matter in the considered case, but the probability of an RTS should be reflected by its occurrence in the set, meaning that the set can include several copies of the same RTS. If the total number of RTSs is small, then all RTSs should be included in the RTS set. If this is not feasible, then the previously found sub-scenarios should be represented by their worst-case RTS in the set. This also applies for the RTSs input to Algorithm 1.

As long as the input sequence of RTSs is representative for real-life system behavior, the algorithm will make sure that the most common RTSs are prioritized in the platform exploitation. If all possible RTSs were included in the RTS set, we now have a complete mapping of all RTSs to their optimal operating points. Each of these operating points and active memory bank combinations make up a scenario for the given RTSs. If groups of RTSs were represented by the worst-case RTSs from the sub-scenario generation, then the rest of the RTSs must be mapped to the same scenario as their worst-case representative. Remaining RTSs can be mapped to the back-up scenario.

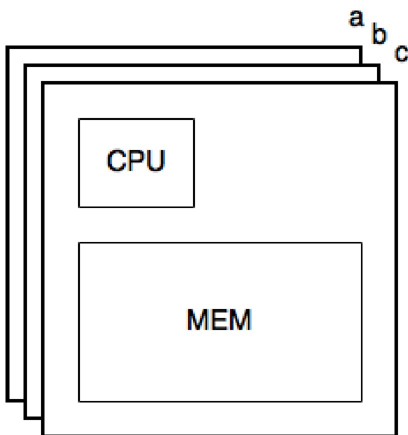
Both CPU and memory energy depend on time. If the available time to execute a task is subject to change at run-time, the inner for-loop in Algorithm 1, which finds the energy of executing at the best operating point (E_{bestOp}) for each RTS in each configuration, should instead calculate a weighted average of all the relevant operating points. The memory configuration $bestConfig$ found in Algorithm 2 will then be the configuration with the best average for all the RTSs. The original inner for-loop in Algorithm 1 can

1 For each memory configuration, map each RTS to the operating point that minimizes total cost. This gives the DVFS sub-scenarios for each memory configuration:

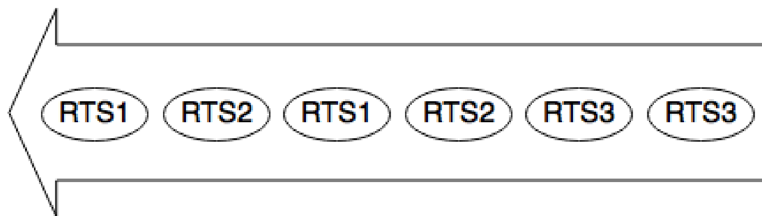
	OP1	OP2	OP3	OP4	OP5	OP6
Config a					RTS1	RTS2, RTS3
Config b				RTS3	RTS1	RTS2
Config c				RTS3	RTS1, RTS2	

2 Find the cost of executing each RTS with the optimal operating point for each platform configuration:

	RTS1	RTS2	RTS3	...
Config a	1680J	987J	509J	
Config b	1701J	998J	460J	
Config c	1558J	732J	478J	



3 For a representative sequence of RTSs, which platform configuration gives the minimum cost?



4 Extract the DVFS sub-scenarios of the optimal configuration (b in this example), and combine this with the memory sub-scenario table of the optimal configuration. The result of this is the scenario look-up table:

	OP1	OP2	OP3	OP4	OP5	OP6
Config a					RTS1	RTS2,RTS3
Config b				RTS3	RTS1	RTS2
Config c				RTS3	RTS1, RTS2	

+

RTS	Active bank combination	Sub-scenario
RTS1	b1	1
RTS2	b1	2
RTS3	b2	2

=

Scenario	RTSs	Active bank combination	CPU operating point
1	RTS1	b1	OP5
2	RTS2	b1	OP6
3	RTS3	b2	OP4

Figure 4.4: Scenario exploration (step 1 and 2), and scenario exploitation (step 3), corresponding to Algorithm 1 and 2 respectively. Step 4 shows how the sub-scenarios are finally combined to form the scenario lookup table.

then be used at run-time (when the available execution time is known) to determine which operating point is optimal. Actual implementations of this is left for future work.

Energy calculations

In this work, the CPU energy is calculated using CPU power characteristics of the Intel Pentium M (see Section 3.1.1, Table 3.2) and Equation 2.2 ($E = P \cdot T$). The memory energy is found using Equation 2.10 along with the CACTI memory characteristics given in Section 3.2. It will be preferable to use energy numbers from actual measurements if this is available.

Calculating the energy consumption of the different memory banks is a complex task. There is a varying number of active banks, with various sizes and energy numbers, and the number of loads and stores depend on the active RTS. Additionally, at this stage of the system development, it is not known exactly which data will be mapped to which memory bank. The rest of this subsection describes how these problems are overcome.

All the memories in a platform configuration contribute to the energy consumption, active or not. To simplify the computations somewhat, the memory banks can either be active or shut off. In active mode, the energy consumption of a bank $bank1$ is then given by this expression:

$$E_{active_{bank1}} = N_{rd_{bank1}} \cdot E_{Read_{bank1}} + N_{wr_{bank1}} \cdot E_{Write_{bank1}} + (T - T_{ShutDown}) \cdot P_{leakActive_{bank1}} \quad (4.1)$$

In shut down mode, the energy is given by:

$$E_{shutDown_{bank1}} = T_{ShutDown} \cdot P_{leak_{ShutDown_{bank1}}} \quad (4.2)$$

These calculations are made for each memory bank and added together to find E_{mem} in line 5 of Algorithm 1.

It is assumed that the memory allocation is evenly distributed among the available memory banks. Furthermore, it is assumed that the number of reads and writes are evenly distributed between all the allocated memory locations. This way, the number of reads and writes to $bank1$, $N_{rd_{bank1}}$ and $N_{wr_{bank1}}$, can be estimated as follows:

$$N_{rd_{bank1}} = N_{rd} \cdot \frac{bank1Size}{totalSize} \quad (4.3)$$

$$N_{wr_{bank1}} = N_{wr} \cdot \frac{bank1Size}{totalSize} \quad (4.4)$$

The total size of the memory configuration is given by *totalSize*, and *bank1Size* is the size of $bank1$.

The assumptions made here will normally not be true. However, with smart compilers, the data that is accessed often can be placed in the smaller memories, where the read energy is significantly lower. By utilizing the smaller memories more, less energy could actually be consumed than the model predicts. The assumptions are therefore not considered unrealistic.

4.4 Prediction

In this step, the RTS prediction mechanisms previously found are now combined and implemented. The RTS parameter values that identify the RTS must be mapped to their scenarios and kept in a lookup table for the run-time prediction. If there are several RTS parameters, the mapping may be implemented as a decision tree, a multidimensional array, or even a combination. In cases where the values of the RTS parameters can only be combined in a few different ways, a decision tree will be better suited than a multidimensional array since such an array can become very sparse. On the other hand, if the RTS parameters can have many different values which can be combined in many different ways, a multidimensional array is probably a better choice than a decision tree, since the decision tree will become very complex.

In cases where the scenario prediction requires much processing of the RTS parameters, it can be considered to pre-process the input data stream before feeding it into the application, if possible. Then the scenario information can be sent along with the stream.

As mentioned, if the available time to execute a task is unknown at design-time, the inner for-loop in Algorithm 1 can be used at run-time to determine the optimal operating point.

4.5 Switching

Switching can get more complex when there are several cost dimensions to exploit. The location and timing of the different reconfiguration procedures can have a considerable impact on the switch delay. This step is very platform dependent, and is not given much focus in this work due to the very theoretical platform that is used.

When the switching mechanisms have been developed, the cost of the switching step can be determined more accurately. The switching costs are determined by the platform and the desired scenario configurations, and usually depend somewhat on which of the scenarios that are switched between. These costs must be taken into account when deciding whether to do a switch or not. The cost of switching scenario should therefore be included in the RTS lookup table. Unless there are big differences in the switching cost, it is okay to assume either the worst-case or average switching cost. To determine whether or not it is feasible to do a scenario switch, the switching cost is compared to the expected energy reduction with the new scenario. If a scenario switch is expected for each RTS, the switching cost can just be added to the total RTS cost. Then the scenario to RTS mapping might have to be updated after this step, in which case the Exploration and Exploitation steps should be repeated.

4.6 Theoretical evaluation of the methodology

The main motivation behind the described methodology is to avoid ending up with sub-optimal combinations of the available system settings. The feasibility of a specific memory configuration for an RTS is heavily influenced by the CPU operating point, and vice versa. Considering all possibilities at the same time can however get very complex, which motivates the changed organization of the first few steps of the methodology. RTS clustering

is not performed based on similarity of cost (as in [6]) but happens as a consequence of optimality with the same operation point. The scenarios are not generated before the platform and application has been explored sufficiently, considering all the dynamism to be exploited. The optimal system knob configuration is thus found.

A problem can arise with the Exploration and Exploitation step if there are settings of one system knob that cannot be combined with the settings of another system knob. For example, if the optimal DVFS operating point of an RTS cannot be combined with a memory configuration that would otherwise be optimal for the RTS. It is then possible to select the second most optimal DVFS operating point or memory configuration instead, but then there is a chance that the resulting combination is sub-optimal.

The non-Pareto optimal CPU operating points are included in the exploration, but not the non-Pareto optimal active memory bank combinations (i.e. variations in the number of active banks for a given configuration). The rest of this paragraph explains why. In the targeted systems, one can not know what will be the optimal operating point for the CPU without including memory costs, e.g. because executing at a lower frequency may increase the memory energy enough to outweigh the saved CPU energy, and vice versa. With the memory configurations however, it is assumed that having as little active memory as possible gives the optimal active bank combination. It is however unknown which configuration will be best suited before including the effects of DVFS. A longer or shorter execution time can determine whether the total system for example benefits from having a special memory available that is optimized for a specific RTS or not.

Chapter 5

Applications

The system scenario methodology is applied to four different C applications, of which the first is a prime number checker developed while studying the system scenario methodology, and the other three are taken from the SPEC CPU™ 2006 integer benchmark suite. All of the considered applications are single-thread implementations.

The three SPEC CPU™ 2006 benchmarks are initially selected because of their assumed dynamic execution times, as will be explained in further detail for each benchmark. All three benchmarks process inputs of different sizes, which by itself can indicate an input-dependent dynamism. Execution time dynamism is confirmed from a few runs with different inputs. SPEC provides three different sets of inputs along with each benchmark; reference, training and test inputs. Reference inputs are used for the reportable results, train inputs provides an input to profile-based compiler optimizations, and the test inputs gives a short run to verify the functionality. All of these are used in the scenario development, as well as some additional inputs in some cases. The benchmarks are run according to the procedure described in Section A.3.

The resulting system scenario designs are presented, along with estimations on energy reduction. All the results are obtained from calculations with the profiling results. The system scenario results are compared to a static implementation, in which the highest CPU operating point is used, and enough memory for any RTSs to be processed is always active.

Assumptions and design decisions

When designing and evaluating system scenarios there are several assumptions and decisions that must be made about the use and requirements for the application. As will be demonstrated, this can have much influence on the final result. The considered applications are all based on processing input data, so it must be known when the processing of each input is expected to start and finish. For example, each input can be processed immediately after the previous input is finished, or periodically. In the latter case, it must also be known what is expected of the CPU and memory when a task is finished. The CPU and memory can for example remain idle or enter a power save mode, depending on the required responsiveness.

The circumstances under which the benchmark applications should be used are mostly unknown, so the system scenarios are developed for two different cases; with and without deadlines. Assumptions for the prime number checker are somewhat different, as will be

explained. The introduced deadlines are long enough for the most demanding input to be processed in the highest performance operating point. When no deadline is assumed, the input processing is continuous. When the deadline is assumed, the tasks have to finish before the deadline. It is also assumed an expected period between each input, in which the CPU and memory have to remain idle while waiting for the next input. When the processing of an input finishes before deadline, the CPU could instead be put in a sleep mode while waiting for the next input. This may however not be feasible in case a more responsive CPU is required. As a compromise between this and leaving the CPU in a high performance state, the CPU's operating frequency will be reduced to 0.6GHz while waiting. Changing the operating point when the processing is finished is not a product of the system scenario methodology, so this is also done in the static implementations that are used for comparison.

5.1 Prime number checker

This application is developed and intended as a starting point for studying the system scenario design methodology. As indicated by its name, the application finds out if the input number is a prime number or not. The application has a limited complexity and is therefore suitable for demonstrating the system scenario concepts. The memory requirements of the application is minimal, so only CPU cycle dynamism is considered. The methodology suggested in Chapter 4 can therefore be simplified somewhat. It is however demonstrated how code transformations can be used to increase the predictability of the scenarios.

The `isPrimeNumber()` function, shown in Listing 5.1, makes up the whole functionality of the application. This function checks whether the number is a prime or not by dividing it on all numbers between itself and two.

Listing 5.1: Function `isPrimeNumber()`

```
bool isPrimeNumber(uint32_t number){
  int i;
  for (i = 2; i < number; i++){
    if (number%i == 0){
      //Number is not prime
      break;
    }
  }
  if (i == number){
    //Number is prime
    return true;
  }
  return false;
}
```

This is a very unsophisticated method, but the worst-case number of iterations of its for-loop structure is easy to find, so that estimates on necessary frequency for a given input number can be made without much overhead. The application must be run with a frequency that lets it finish within the deadline even for a prime number, so the size of

the input number gives an upper limit on the number of iterations necessary to reach a conclusion.

The possible inputs are any random number between 1 and 637,538,053, and the maximum available time for each input is 12 seconds. In the first case to be considered, a new number is input immediately after the processing of a number has finished. The next case is with a fixed interval between each input.

5.1.1 DVFS dynamism

RTS identification

The CPU cycle count of the application is profiled using an set of pseudorandom, evenly spaced numbers from 1 to 637,538,053 as input. Much execution time dynamism is identified from this, and from code inspection it can be seen that there are two sources of dynamism. The size of the number is proportional to the worst-case number of iterations of the while loop. This is a dynamism that can be taken advantage of without much run-time overhead, as the size of the input number can be used as the RTS parameter. It is however often sufficient with a much smaller number of iterations than the worst-case estimation; e.g., every second number is dividable by two, and every third number by three. This means that $50\% + 0.5 \cdot 33.33\% = 66.66\%$ of all decisions will finish within two iterations. The actual needed number of iterations before the function returns makes up a dynamism which is harder to exploit.

The function `isPrimeNumber()` is responsible for almost all the CPU cycles, as the processing happens here. This function can be partitioned by looking at each while iteration. Through profiling it is observed that the number of clock cycles per iteration is the same regardless of the input size. The loop body can thus be considered a black box, or thread node. The modulo operation is responsible for most of the CPU cycles. No further decomposition of the code is necessary.

In Figure 5.1 the clear relationship between the size of the number and worst-case required CPU cycles can be seen. Obviously, all the 637,538,053 RTSs cannot be profiled. The linearity of the worst-case CPU cycle count is therefore used to create an expression that returns the worst-case CPU cycles given any number between 1 and 637,538,053. The expression is found using linear regression with the profiling results, shown in Figure 5.1.

RTS prediction

The input number is used as the RTS parameter, and the worst-case number of clock cycles for different input numbers will be estimated using the relationship shown in Figure 5.1. The CPU frequency requirements will be calculated from this, giving an initial RTS to scenario mapping.

DVFS sub-scenarios

If assuming a deadline of 12 seconds for each input number, the biggest prime number (637,538,053) will finish within the deadline if executing at 1.6GHz. For the smaller prime numbers, lower frequencies may be enough. Each input number gives a different

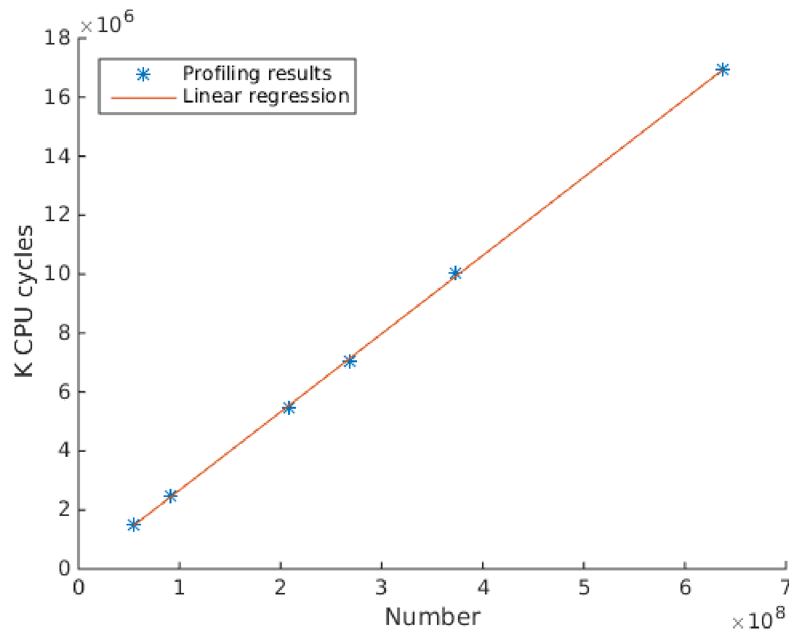


Figure 5.1: K CPU cycles when running the prime number checker with six random primes, and the linear regression of these measurements ($Kcycles = 19369.929190503 + 0.026529737759965 \cdot number$).

RTS, so all the RTSs cannot be mapped directly to a sub-scenario. Each sub-scenario should instead contain a group of consecutive numbers (represented by the maximum number) that can be executed with that sub-scenario without breaking the deadline. The expression found during the RTS identification (Figure 5.1, Equation 5.1) is used to find the maximum number for each operating point.

$$cycles = 19369929.190503 + 26.529737759965 \cdot inputNumber \tag{5.1}$$

An expression that gives the minimum CPU operating frequency for a number is found as follows:

$$\begin{aligned} minFrequency &= \frac{cycles}{maxTime} && (5.2) \\ &= \frac{19369929.190503 + 26.529737759965 \cdot inputNumber}{12\text{s}} && (5.3) \\ &= (1614161 + 2.2108 \cdot inputNumber)\ \text{Hz} && (5.4) \end{aligned}$$

By reordering this expression we get the maximum input prime number as a function of the frequency:

$$inputNumber = (frequency - 1614161)/2.2108 \tag{5.5}$$

From Equation 5.5, the RTS table (Table 5.1) is generated, giving the worst-case input that can be processed at each available operating point. Sub-scenario 1 is used as the backup sub-scenario.

Table 5.1: Mapping of worst-case RTS parameter to DVFS sub-scenarios.

Worst-case RTS parameter	**Minimum operating frequency**	**Sub-scenario**
722989795	1.6GHz	1
632524805	1.4GHz	2
542059814	1.2GHz	3
451594824	1.0GHz	4
361129834	0.8GHz	5
270664844	0.6GHz	6

5.1.2 Memory dynamism

Very little memory is used by this application, and the memory dynamism is minimal, so no memory scenarios will be developed for this application.

5.1.3 Exploration and Exploitation

No other costs than CPU power are considered in the scenario generation, since the memory energy consumption should be minimal. Therefore, due to the Pareto-optimality of all the CPUs operating points except 0.6GHz (see Section 3.1), it can be assumed that the sub-scenario with the lowest possible operating frequency from 0.8GHz and up will be optimal for the system. Similar to Table 5.1, Table 5.2 gives the maximum number (RTS parameter) that can be processed in each scenario. Sub-scenario 6 is however removed because as was not Pareto-optimal.

Table 5.2: Mapping of worst-case RTS parameter to scenarios.

Worst-case RTS parameter	**Operating Point**	**Scenario**
722989795	1.6GHz	1
632524805	1.4GHz	2
542059814	1.2GHz	3
451594824	1.0GHz	4
361129834	0.8GHz	5

The mapping in Table 5.2 ignores the fact that some cycles will be lost to prediction overhead and switching delays. This is however very platform dependent, so only a short description of how this can be incorporated is given in the next section.

5.1.4 Prediction and Switching

Two possible strategies are considered for the prediction. Equation 5.4 can be used at run-time to calculate the minimum frequency f_{min} required for a given input number, assuming it is a prime number. Then the scenario corresponding to this f_{min} can be found from a lookup table. This is an effective implementation in terms of memory, but not time. Run-time overhead can be saved by doing these calculations at design-time instead, and mapping only the worst-case RTS to each scenario. Time is more critical than memory in this application, so the latter solution is chosen.

The optimal scenario for each number range is therefore taken from Table 5.2 and stored in a lookup table. The predictor uses this table at run-time to look up the ideal scenario for each input number. The scenario predictor function (AMU, see Section 3.3.1) is placed at the beginning of `isPrimeNumber()`, so that before any input processing begins, it is checked which scenario is optimal. The scenario switching function (PAM, see Section 3.3.1) checks whether the current scenario is different from the optimal scenario, and then switches to the optimal scenario if this is the case. The function is called right after the scenario predictor has found the optimal scenario.

It is however observed that the scenario switches happen very frequently with this implementation. Many of the input numbers are dividable by some of the first primes (e.g., 2, 3, 5, 7, 11), which means that very few iterations of the for-loop shown in Listing 5.1 will be executed. In these cases the scenario should not be changed, as the gain of doing so is probably outweighed by the overhead caused. With random input numbers it cannot be assumed that the next number will belong in the same scenario as the current. To prevent the short scenario periods, the first iterations of the for-loop are unrolled and executed *before* the scenario functions are called.

The number of iterations of the for-loop that should be executed before a scenario switch is considered depends on the switching delay, and is therefore very platform dependent. However, the exact value of this limit is not very important, as long as it is more than 3 (the majority of all numbers are dividable by 2 and/or 3 anyway). Figure 5.2 illustrates this by grouping all numbers between 1 and 10000 according to their smallest factors. E.g., the first bin contains all numbers that are dividable with a number between 2 and 10. The next bin contains the numbers that are not dividable with numbers between 2 and 10, but with numbers between 11 and 100.

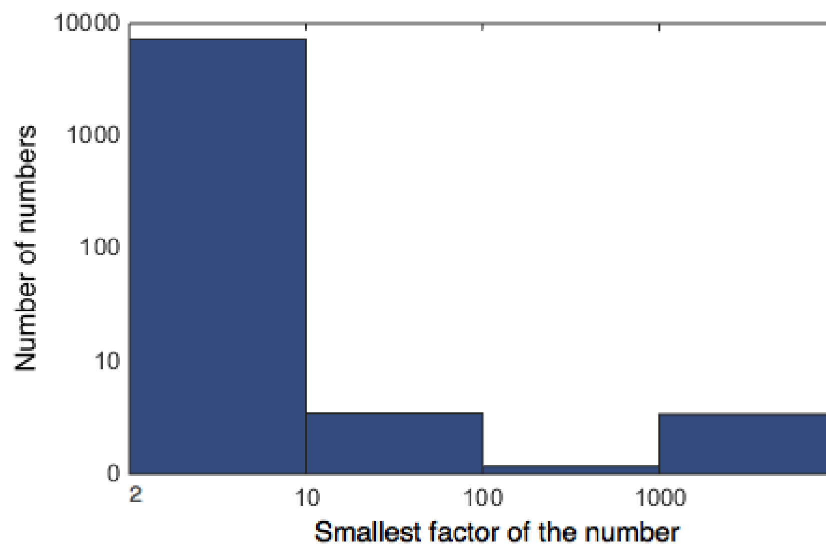


Figure 5.2: The number of numbers that have their smallest factor either between 2-10, 10-100, 100-1000 or 1000-10000. Note that both axis are shown logarithmically.

By delaying the scenario prediction, the probability of predicting the right scenario is increased drastically. As shown in Section A.2 in the Appendix, by checking if the input number is dividable by two or three before predicting scenario, the probability for

scenario misprediction is reduced from 95.1% to 28%:

$$0.951 - 0.50 - 0.50 \cdot 0.33 = 0.28 \tag{5.6}$$

Delayed scenario prediction can be considered for applications that are similar to the prime number checker, i.e. when a loop or function is responsible for most of the execution time, and it can potentially exit very early due to some condition which is unknown before starting the loop or function. More generally, the scenario prediction in input processing applications can improve from this if the processing of some inputs are aborted during the initial processing. Characteristics of such applications are difficult to generalize. The potential can however be recognized through profiling and code inspection. Anyway, if the scenario misprediction rate is high for a system scenario implementation, the benefit of delaying the prediction should be investigated.

5.1.5 Results

The application is profiled again, both with and without system scenarios, with a new sequence of 100 random numbers between 0 and 637,538,053. The number of scenario switches and CPU cycles within each scenario is recorded and the CPU energy is calculated from this. The scenario implementation is estimated to use 538.6J, and the original implementation 674.6J, meaning that 20.16% CPU energy is saved with the scenario version. This is assuming continuous input processing.

Profiling shows that the CPU cycles of the original code compared to the partly unrolled `isPrimeNumber()` with the scenario functions is only increased by 0.43%. Adding the 10us CPU unavailability time per CPU operating point switch (18 switches for the random sequence of 100 numbers) does not make any notable difference.

5.1.6 Changed assumptions

The assumptions for the scenario development is now slightly changed; it is now assumed that the time between each new input number is expected to 12 seconds. Whereas the processing of each number before started immediately after the previous number was finished, there is now some expected waiting time between each input. This changes the scenario set and prediction/switching somewhat. First of all, a scenario for the waiting time is needed. The operating point with the lowest power consumption is selected for this. In order to not perform unnecessarily many switches, this scenario will also be used for the first few iterations of each input number. Otherwise, the scenario implementation is as before. The implementation is profiled as before, with a sequence of 100 random numbers. The results are first compared to a static (race to idle) implementation, using the same operating point (1.6GHz) the whole time. The calculated CPU energy consumption is 7451J for the scenario implementation, and 29400J for the static implementation. This shows an energy reduction of 75%. In conclusion, dynamic applications like the prime number checker represent a promising domain for system scenario development when subjects to such execution conditions.

As argued in the beginning of this chapter, the static implementation should however be able to operate with minimum performance while waiting for the next input to arrive. If this is assumed, the static implementation's CPU energy consumption is reduced to 7709J,

and the scenario implementation uses only 3.3% less than the static implementation. The scenario implementation with continuous input processing used only 538.6J, so when 12 seconds between each input is assumed it is clear that most of the energy is spent waiting. This can explain why only 3.3% energy is saved with this assumption.

5.2 456.hmmer

The HMMER package is a set of tools, mostly used in computational biology to align and search for protein and gene sequences in databases [16]. HMMER uses profile-Hidden Markov Models (HMMs) to represent the alignment. A profile-HMM is a statistical model of a gene sequence, consisting of a series of nodes where each node correspond to a position (or column) in the alignment from which it was made. The number of nodes is also referred to as the length of the profile-HMM, or the *model length*. With a profile-HMM, one can effectively search through huge databases of gene sequences to find a matching sequence, or members of the protein sequence family. More information about this can be found in the HMMER User's Guide [45].

The HMMER benchmark runs two methods (`hmmercalibrate` and `hmmersearch`) from the HMMER package. When only a profile-HMM is given as input to the application, `hmmercalibrate` is executed. This method calibrates the input profile by generating a large number of random sequences which are then compared to the original profile. When the calibrated profile-HMM is later used to search a database, the search will be more sensitive due to the calibration. To run the other method, `hmmersearch`, the benchmark application needs two inputs; a profile-HMM and a gene sequence database. The application then searches the database to find related sequences, and returns a ranked list of best-matches. `hmmersearch` is the method that will be used to evaluate the system scenario methodology. Four profile-HMMs and a database are provided as input to the benchmark. The provided profile-HMMs make up the entire considered input set. Judging by the size of these inputs (from 10 to 300 nodes) and the fact that these will be compared to sequences in a database, it is reasonable to believe that there will be some input-dependent dynamism in this application. The application uses the Viterbi algorithm to do the search, which has previously been used in a system scenario design [13]. Other possible use cases of HMM and the Viterbi algorithm are numerous [46] (some examples are given in Section A.5), which further motivates using this benchmark to demonstrate and evaluate the system scenario methodology.

5.2.1 DVFS dynamism

This section describes some of the CPU cycle dynamism in `hmmersearch` and how it is found.

RTS identification

A few runs of the benchmark with the different profile-HMMs and the same database confirms the expected dynamism due to different model lengths (i.e. number of nodes). The model length is denoted as `hmm->M` in the HMM struct. This parameter is known from the start, so it is a suitable RTS parameter. Each of the provided HMM-profiles

gives an RTS; `bombesin.hmm` (RTS 1), `leng100.hmm` (RTS 2), `retro.hmm` (RTS 3) and `nph3.hmm` (RTS 4), with respective model lengths of 10, 100, 100 and 300 nodes. The provided database, `swiss41` consists of 122564 sequences.

Using the procedure described in Section 4.2.1, the application is decomposed as shown in Figure 5.3, 5.4 and 5.5. The function `P7Viterbi()` is responsible for around 98-99% of the CPU cycles. The rest of the application can therefore be modelled as black-boxes. Figure 5.5 illustrates where the dynamism comes from in `P7Viterbi()`.

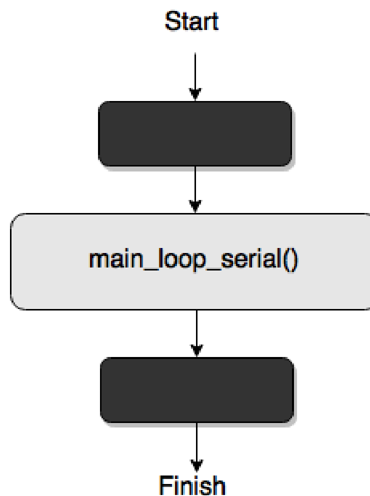


Figure 5.3: `hmmersearch` top-level model.

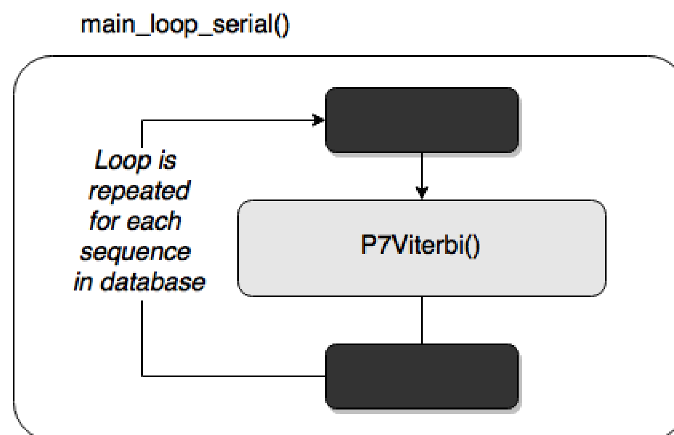


Figure 5.4: Gray-box model of the main function.

The following potential RTS parameters are identified:

- `DB_size`: Number of sequences in the database. Can be found in advance by iterating through the database.
- `L`: Length of a sequence in the database. The length of the sequences in `swiss41` varies a lot.
- `M`: Length of the profile-HMM. Found in the description field of the input HMM file.

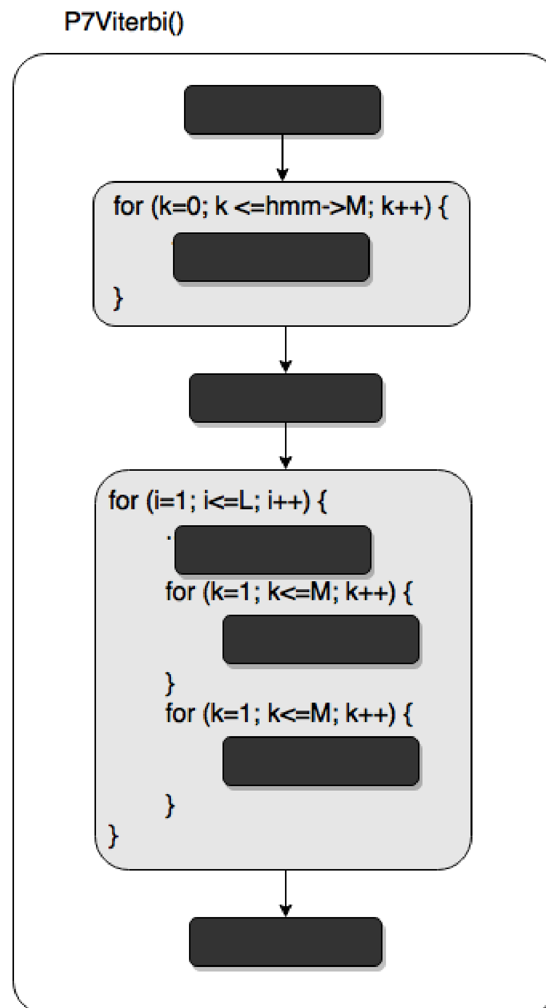


Figure 5.5: Gray-box model of the `P7Viterbi()` function.

`P7Viterbi()` is called once for every sequence in the gene database, i.e. `DB_size` number of times (Figure 5.4). This size is kept constant at 122564 (the size of the `swiss41` database). 90-99% (depending on the model size) of the cycles spent on executing `P7Viterbi()` are due to the for-loops shown in Figure 5.5. The execution time of each `P7Viterbi()` run depends on the input profile-HMM and the length of the currently considered database sequence.

In order to test whether the size of the input database affects the execution time, the database is split into smaller databases that are used instead of the original. As expected from the gray-box model in Figure 5.4, it is observed that the size of the input database (i.e. the number of sequences in the database) does affect the execution time. For the tested database sizes, the execution time decreases linearly with reduced databases. Since only one database is available however, this dynamism will not be further explored in this work. A constant database is assumed.

RTS prediction

The model length will be used as the RTS parameter. As mentioned, the four profile-HMMs used in the benchmark suite have three different model lengths; 10, 100 and 300. If it can be assumed that most profile-HMMs are of these sizes, the similar RTSs can simply be recognized from this.

If the possible input is not limited to these number of nodes, an expression can be found (as with the prime number checker) that gives the number of CPU cycles from the RTS parameter(s). The expression in Equation 5.7 is found by profiling the application several times with different inputs in Zoom. The number of required CPU cycles for each black box of the `P7Viterbi()` function (see Figure 5.5) are found by looking at the cycle count line annotations at assembly code level. These cycle counts are fairly constant independent of input. Equation 5.7 is then found as described in Section 4.2.1.

$$\text{CPU cycles P7Viterbi()} = (M+1)\cdot 1737 + M\cdot 2416667 + M\cdot 193333 + 608148000 \quad (5.7)$$

This expression gives the total number of CPU cycles spent in `P7Viterbi()` for a whole application run when the `swiss41` database is used. The dependency on model size `M` is clear and in correspondence with Figure 5.5. The cycles outside any of the M-dependent loops in `P7Viterbi()` are contained in the constant at the end of the expression. If different databases are to be used, the size of the database and the length of its sequences can also be incorporated to the model. The accuracy of the model can be seen in Figure 5.6.

If we assume that there can be other model lengths than 10, 100 and 300 (and that Equation 5.7 also holds for these), this expression can be used to implement system scenarios for any model length. This is similar to what is already demonstrated with the prime number checker in Section 5.1. The longest profile-HMMs that should be executed at each CPU operating point to not break any deadlines can then be pre-calculated at design-time, so that when a new input is received, the suitable CPU frequency is found by looking up the model length in the RTS table. This is however left for future work, as only the four RTSs from the four provided inputs will be considered in the scenario exploration.

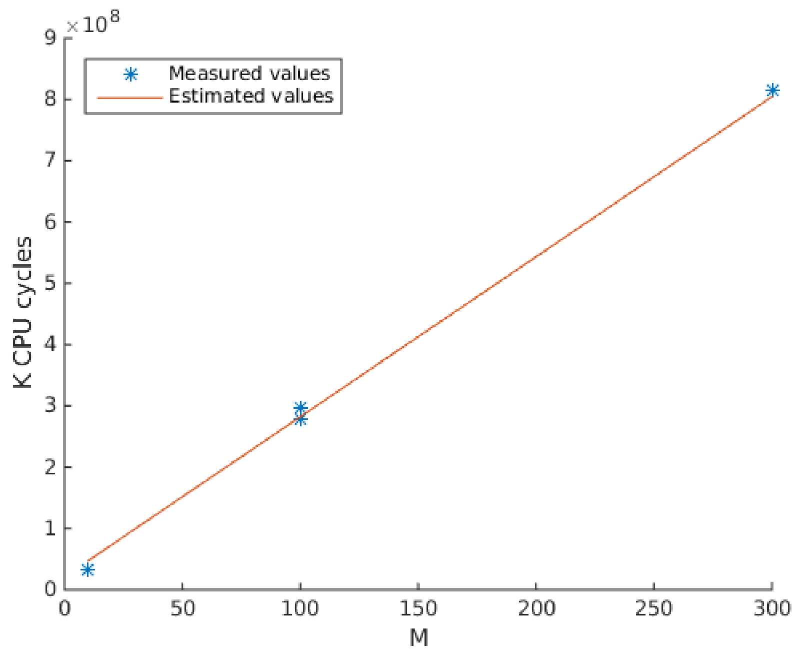


Figure 5.6: `P7Viterbi()` CPU cycles.

DVFS sub-scenarios

The RTSs can be grouped by the model length, as there is a clear relationship between the model length and the required number of CPU cycles. Judging by the CPU cycle counts of the two inputs with 100 nodes (shown in Table 5.3) there is not much variation in cycle count for the same model lengths. Initially, no deadlines are assumed for this application. This means that all operating points can be used for each RTS, i.e. all RTSs are part of the same sub-scenario where all operating points are allowed. If however a deadline is assumed, e.g. of 550 seconds, which is just enough for the worst-case RTS to finish at 1.6GHz operation, we get the sub-scenario table shown in Table 5.4.

Table 5.3: CPU cycles when executing `hmmersearch` with database `swiss41` and the given inputs.

RTS	Input	Model length	K CPU cycles
1	bombesin.hmm	10	33287000
2	leng100.hmm	100	277735000
3	retro.hmm	100	296593000
4	nph3.hmm	300	814576000

Table 5.4: DVFS sub-scenarios when a deadline of 550 seconds is assumed

Worst-case RTS parameter	Min. operating frequency	Sub-scenario
100	0.6GHz	1
300	1.6GHz	2

5.2.2 Memory dynamism

RTS identification

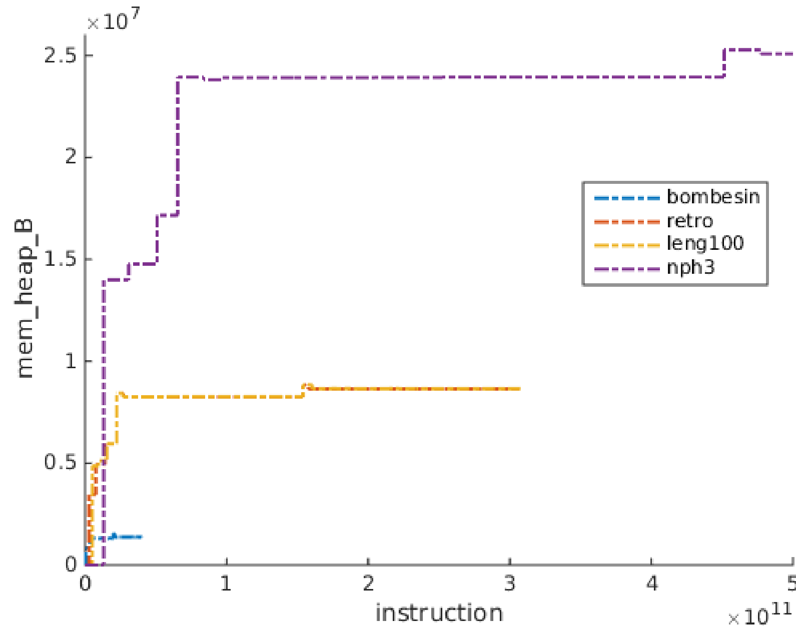


Figure 5.7: Memory allocation in `hmmersearch`. Allocated memory size is very input-dependent. The right side of the graph has been truncated in order to see the details to the left better.

Running the application with Massif reveals that `ResizePlan7Matrix()` is responsible for almost all memory allocation. The memory profiling results are given in Figure 5.7, showing enormous variations in allocated memory for `hmmersearch`. The full memory allocation occurs early in the application, making it feasible to exploit the memory space dynamism. Again, it is the length of the input profile-HMM that determines how much memory is allocated (verified with code inspection), so this should still be the RTS parameter. There is no reuse of memory from one input to another. Even though the database size will be kept constant in this system scenario design, it is investigated how it affects the amount of allocated memory. This is tested by searching for `bombesin.hmm` in the full database, half the database, and a quarter of the database. Figure 5.8 shows that the allocated memory is actually independent of the size of the input database. This is because the application never allocates memory for the whole database, but reads it sequentially from the database file.

Table 5.5 gives a summary of the relevant memory use characteristics of `hmmersearch`, obtained from Massif and Cachegrind according to the procedure described in Subsection 4.2.2.

RTS prediction

The memory RTSs can be recognized exactly the same way as the DVFS RTSs, since the same RTS parameter is used. At the beginning of each run, the necessary memory size

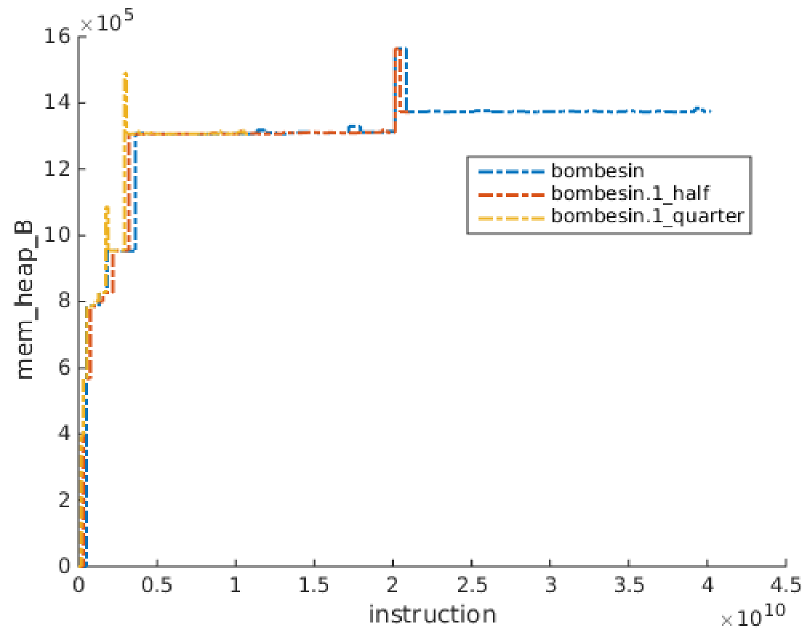


Figure 5.8: Memory allocation in HMMER with different database size. Size of allocated memory does not depend on input database.

Table 5.5: Memory use characteristics of `hmmersearch` for each input.

RTS	Model length	Max. allocated mem.	Loads	Stores
1	10	1.4MB	17922053766	5973689604
2	100	8.6MB	135438406314	46137092046
3	100	8.7MB	135300405900	46171892568
4	300	25.0MB	396805190412	136468446996

can be found from a lookup table and the excess memory can be powered down.

Memory sub-scenarios

Considering the required memory shown in Table 5.5 and the available sizes presented in Section 3.2, the following memory bank configurations are suggested:

a) 32MB

b) 32MB + 2MB

c) 32MB + 8MB + 2MB

d) 16MB + 8MB + 1MB

e) 16MB + 8MB + 2MB

f) 16MB + 8MB + 2MB + 1MB

With each configuration, the minimal amount of memory is enabled for each RTS. These active bank combinations are the memory sub-scenarios of each memory configuration. Up to three different memory sub-scenarios, each providing 2MB (RTS 1), 9MB (RTS 2 and 3) and 25MB (RTS 4), are used for each memory configuration.

5.2.3 Exploration and Exploitation

It is assumed that the probability distribution of the input model length is reflected in the available input set. The scenarios are thus optimized based on a sequence consisting of the four profile-HMMs of length 10, 100, 100 and 300, i.e. E(10) + E(100) + E(100) + E(300) is minimized. Initially, no deadline is assumed. Then a deadline of 550 seconds per input is assumed, which is just enough for RTS4 to finish at maximum performance. By using Algorithm 1 and 2, the optimal combination of the DVFS and memory sub-scenarios are found for all RTSs. The different alternatives and the optimal combinations are shown in Figure 5.9 without deadline, and in Figure 5.10 with deadline.

From Figure 5.9 and 5.10 the optimal memory platform configurations when all RTSs are considered can be seen. Without deadline, the optimal configuration is d) 16MB + 8MB + 1MB, with the CPU operating at either 1.0GHz or 1.2GHz. The differences in energy consumption are however quite subtle. When assuming a deadline of 550 seconds per input, configuration e) is slightly more energy efficient. The optimal operating point of RTS 1, 2 and 3 are now 0.6GHz, while RTS 4 has to be executed at 1.6GHz in order to finish.

The following list describes the obtained scenario set when no deadline is assumed:

- Scenario 1 (RTS 1): 16MB shut off, 8MB active, 1MB shut off, CPU executing at 1.0GHz
- Scenario 2 (RTS 2 and 3): 16MB shut off, 8MB active, 1MB active, CPU executing at 1.0GHz

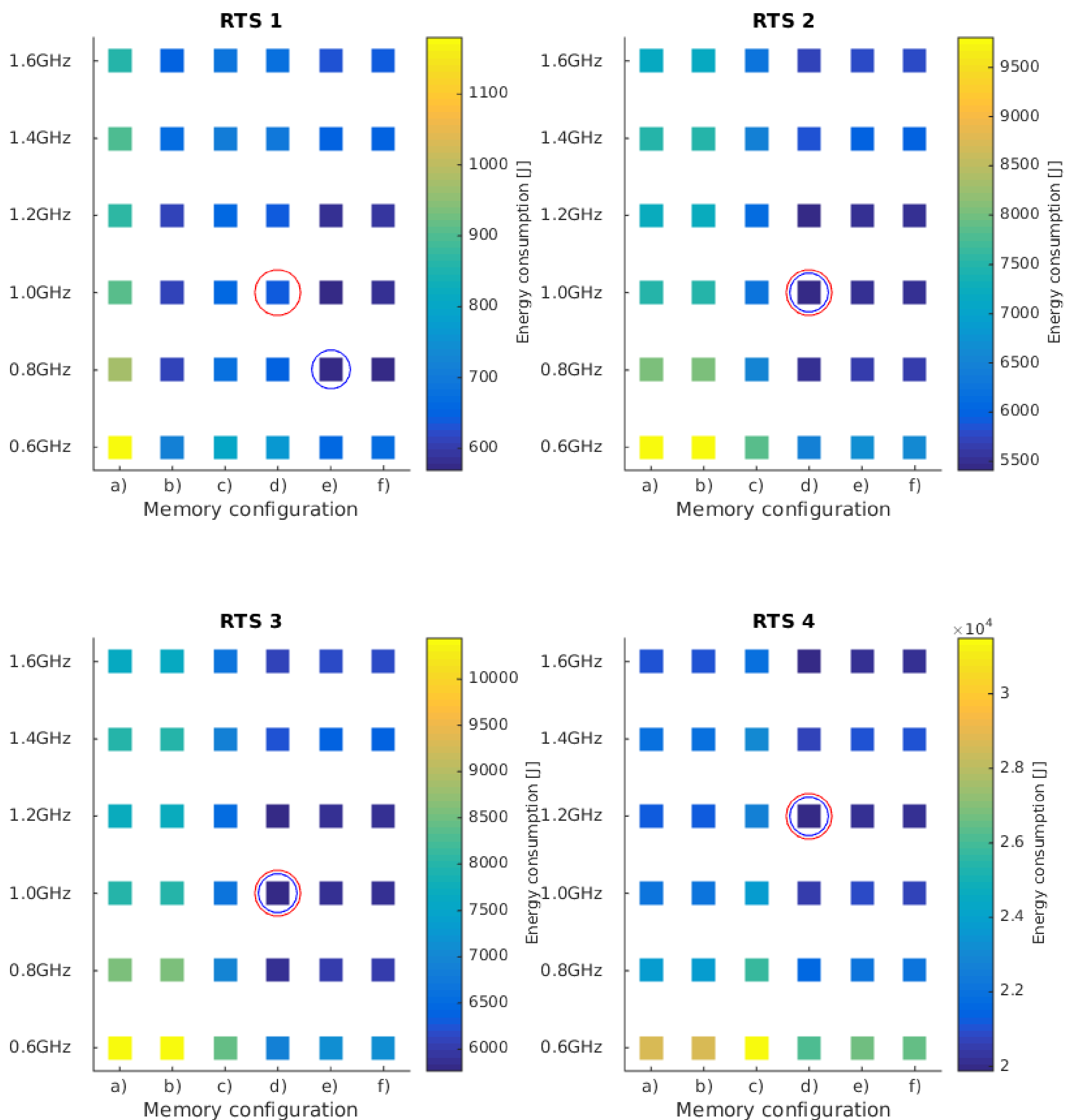


Figure 5.9: Different energy consumption of each RTS depending on CPU operating point, denoted by frequency on the y-axis, and the memory configurations along the x-axis. The blue circle marks the optimal combination of CPU operating point and memory configuration for each RTS individually, and the red circle marks the best CPU operating point for each RTS given the best memory configuration for all the RTSs combined.

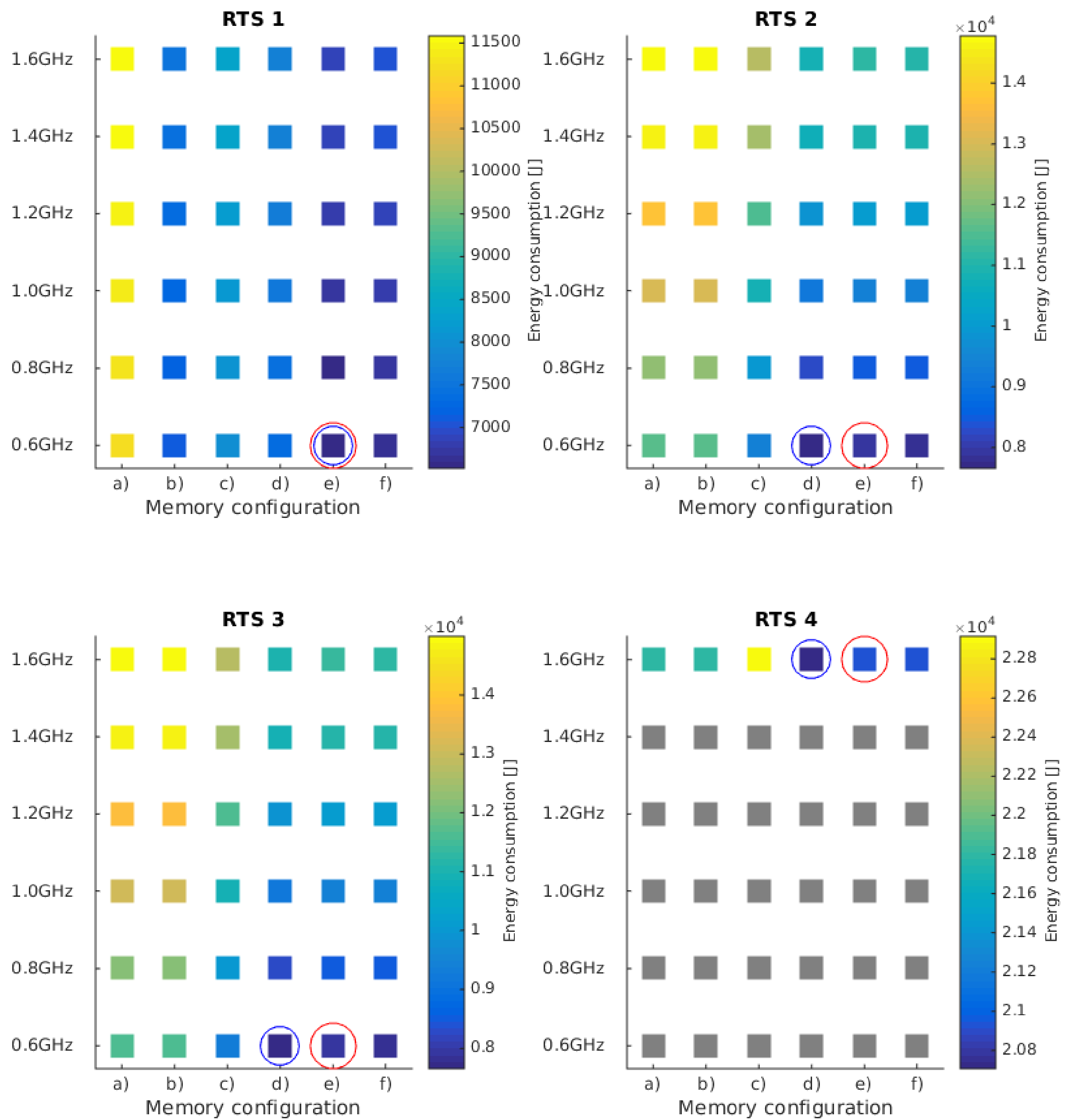


Figure 5.10: Different energy consumption of each RTS depending on CPU operating point, denoted by frequency on the y-axis, and the memory configurations along the x-axis. The blue circle marks the optimal combination of CPU operating point and memory configuration for each RTS individually, and the red circle marks the best CPU operating point for each RTS given the best memory configuration for all the RTSs combined. Operating points that will break the deadline are colored gray.

- Scenario 3 (RTS 4): 16MB active, 8MB active, 1MB active, CPU executing at 1.2GHz
- Backup scenario: 16MB active, 8MB active, 1MB active, CPU executing at 1.6GHz

When the deadline is assumed, the following scenarios set is obtained:

- Scenario 1 (RTS 1): 16MB and 8MB shut off, 2MB active, CPU executing at 0.6GHz
- Scenario 2 (RTS 2 and 3): 16MB shut off, 8MB and 2MB active, CPU executing at 0.6GHz
- Scenario 3 (RTS 4): 16MB, 8MB and 2MB active, CPU executing at 1.6GHz
- Backup scenario: 16MB, 8MB and 2MB active, CPU executing at 1.6GHz

5.2.4 Prediction and Switching

Combining the RTS prediction mechanisms found earlier is straightforward, since the same RTS parameter is used and the memory sub-scenarios correspond one to one with the DVFS sub-scenarios. The scenarios will therefore be predicted from the number of nodes in the input.

A mechanism that configures the system according to the scenario mapped to the input model length must be implemented, as the scenario manager described in Section 3.3. The CPU unavailability time of 10µs per change of operating point can be neglected when compared to the average execution time. So can the additional CPU cycles of the scenario management. Memory banks are activated once during execution, so the wake-up energy for each memory can also be neglected. The following list formulates the scenario prediction:

- For input with 10 nodes or less, use scenario 1.
- For input with 100 nodes or less, use scenario 2.
- For input with 300 nodes or less, use scenario 3.
- Otherwise, use the backup scenario.

5.2.5 Results

Figure 5.11 and 5.12 compares the total energy consumption of the different memory configurations for the whole sequence of RTSs, without and with deadline respectively. The leftmost points are of the static implementation using the highest CPU frequency and a 32MB memory bank regardless of RTS. Without deadline, the total energy reduction with the optimal scenario implementation compared to the static implementation for the whole RTS set is only 13.4%. This is mainly because RTS 4, which is the most time demanding RTS, is relatively energy efficient in the static implementation (see Figure 5.9). With deadline, the total energy reduction is 31.2%. This is under the assumption that a new input arrives at each deadline. When the processing of an input is finished before the deadline, the operating point with the lowest power consumption (0.6GHz) is

used until the next input arrives, both for the scenario implementation and the static implementation.

Figure 5.13 and 5.14 show how much the energy of each RTS is reduced without and with deadline. The energy contribution of CPU and memory is separated so that it can be seen where the reductions take place. Both with and without deadline it can be seen that the energy reduction in RTS 4 is minimal. It can also be seen that without deadline, the energy distribution is quite similar with and without scenarios. The energy reduction is however substantial when the deadline is assumed. Much of the CPU energy reduction is because the low performance operating points used for RTS 1, 2 and 3 reduces the number of CPU cycles between each input. The memory power reduction from utilizing memory-aware scenarios becomes more evident for these RTSs because of the idle times between these inputs.

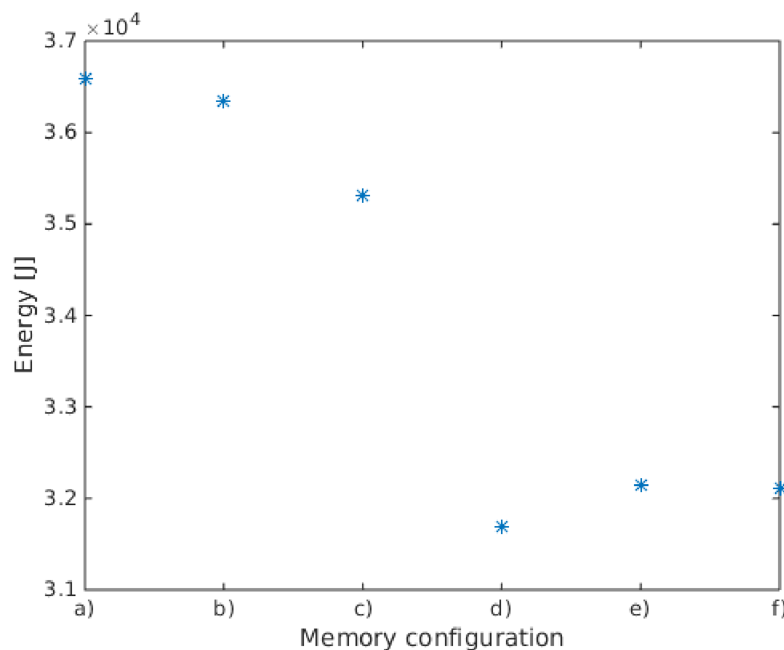


Figure 5.11: Total energy of all the RTSs for each of the different memory configurations. Each RTS is executed in its optimal operating point.

5.2.6 Other observations

The application has still more potential for system scenario design, which should be further explored. Single-thread execution is the default, but both `hmmercalibrate` and `hmmersearch` can be multi-threaded by adding some `#defines`. `P7Viterbi()` function calls can then be carried out in parallel, and most of the execution becomes parallelized. With `hmmercalibrate` the input sequence is compared to randomly generated sequences in parallel, and with `hmmersearch` the input sequence is compared to database sequences in parallel. The speed increases almost linearly with the number of processors used, given ideal conditions [47]. This makes the application very suited for system scenario design with multiple processing elements.

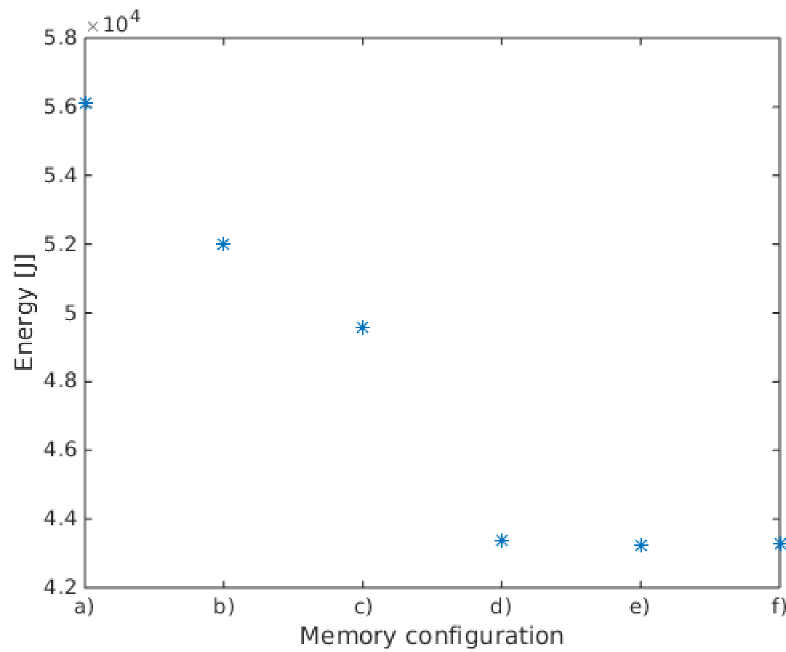


Figure 5.12: Total energy of all the RTSs for each of the different memory configurations, with deadline. Each RTS is executed in its optimal operating point.

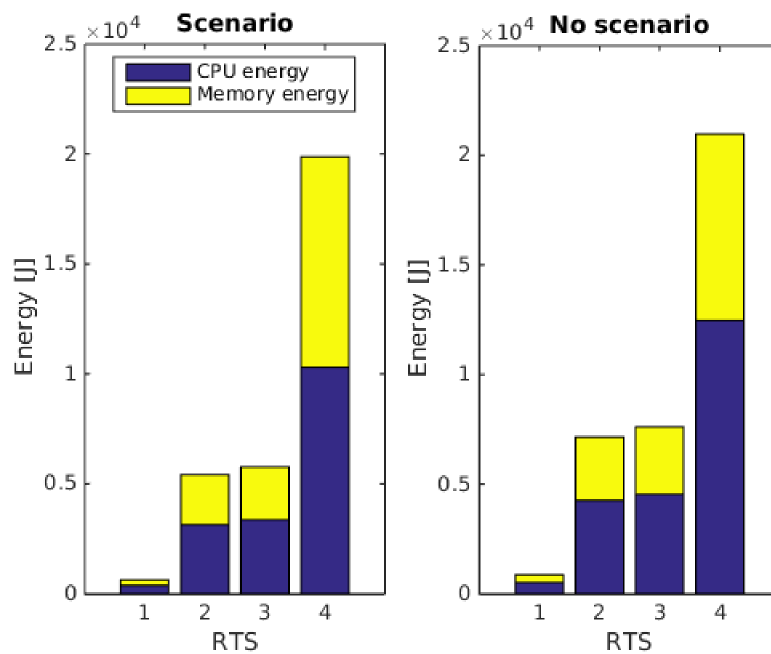


Figure 5.13: Comparison of the total energy consumption of the static, no scenario implementation versus the optimal system scenario implementation for each RTS. No deadline.

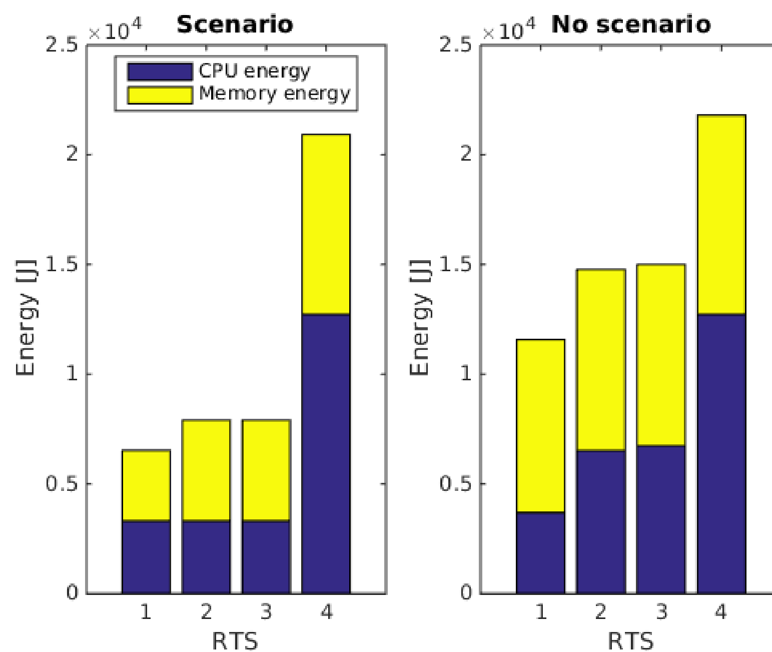


Figure 5.14: Comparison of the total energy consumption of the static, no scenario implementation versus the optimal system scenario implementation for each RTS. With deadline.

The parallel nature of the application can also be exploited by allowing simultaneous memory accesses through memory bank partitioning. The access pattern is however very sequential, causing very few cache misses anyway, so a simple pre-fetching strategy should also be very efficient.

5.3 429.mcf

This benchmark is derived from MCF, a program that solves single-depot vehicle scheduling problems in public mass transportation. These are solved as Minimum Cost Flow (MCF) problems, using an implementation of the network simplex algorithm accelerated with column generation [16].

The input to the application is a text file with a list of timetabled trips with fixed times and locations for departures and arrivals. The rest of the text file consist of the links between the timetabled trips, so-called dead-head trips, as well as pull-in and pull-out trips for entering and leaving the depot. Costs are given for all trips except the timetabled. Each timetabled trip must be served by exactly one vehicle, and the vehicle fleet is homogeneous. The timetabled trips are to be scheduled so that the cost is minimized, and so that the number of necessary vehicles is as small as possible.

According to the benchmark description given in [16], the worst case execution time is *"pseudo-polynomial in the number timetabled and dead-head trips and in the amount of the maximal cost coefficient"*. The expected execution time of the application is a low-order polynomial. Judging from this, there should be some input-dependent dynamism, but an accurate execution time predictor can possibly be difficult to obtain.

5.3.1 DVFS dynamism

RTS identification

Running the application with the provided input set gives very varying execution times, so there is clearly much CPU cycle dynamism in this application. Roughly, more nodes (time-tabled trips) requires more cycles to process. A brief look at the code reveals many for and while loops. In addition to the provided input set, an MCF input generator [48] (see Appendix A.4) is used to create inputs of various sizes. The maximal cost coefficient could however not be controlled with this generator, and its effect on the execution time is therefore suggested for future work. The dynamism due to different node number is confirmed by running the application with the additional input set. By profiling the application in Zoom, it is noted that the functions `primal_net_simplex()` and `price_out_impl()` are responsible for 31.8% and 33.0% of the total CPU cycle count when the reference input is used. The cycle count of these functions vary a lot with different input sizes. The cycle count of the other functions vary little, and are therefore modelled as black-boxes. The top-level structure of the application is shown in Figure 5.15.

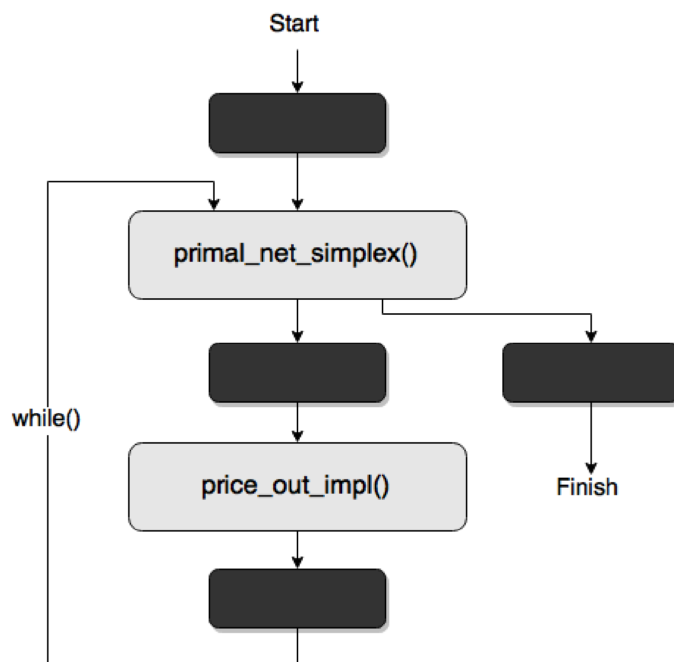


Figure 5.15: Gray-box model showing the two functions that are responsible for most of the CPU cycles.

The loop seen in Figure 5.15 is a while loop which is always executed five times, due to a define directive. The functions `primal_net_simplex()` and `price_out_impl()` are thus repeated six and five times respectively. To further investigate the dynamism (as described in in Subsection 4.2.1), each of these function calls are profiled separately. The amount of CPU cycles required by each call seems to depend on the number of input nodes and on which iteration is being executed, as seen in Figure 5.16 and 5.17.

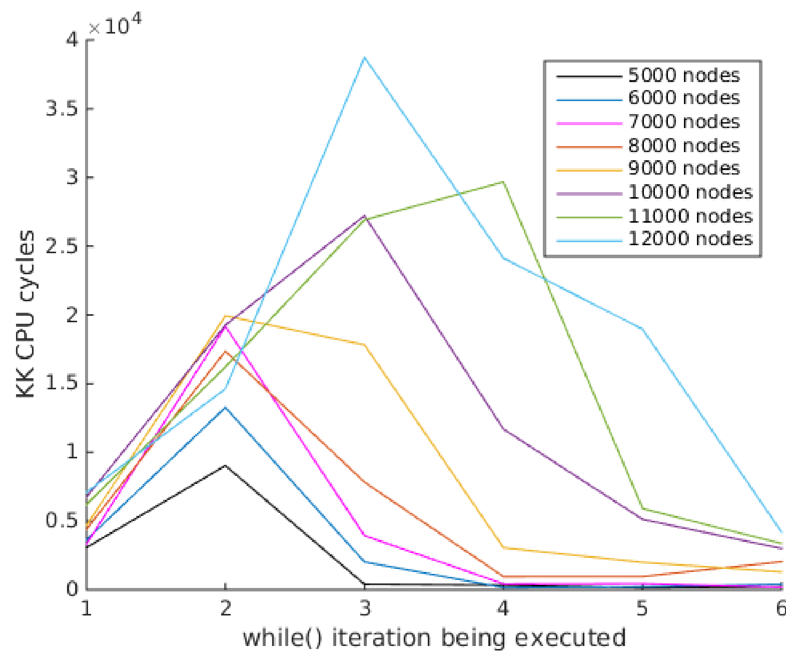


Figure 5.16: The number of CPU cycles spent on executing `primal_net_simplex()` is very dependent on how many times it has already been executed.

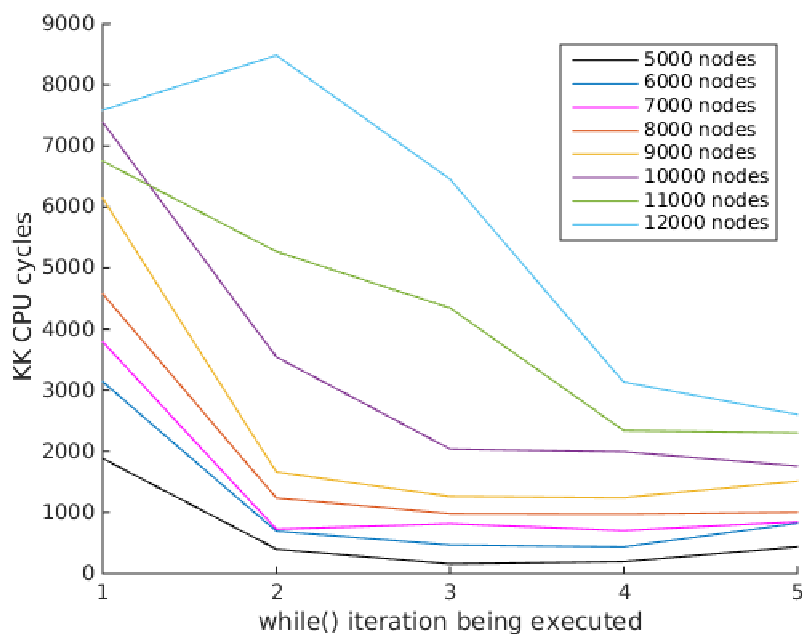


Figure 5.17: The number of CPU cycles spent on executing `price_out_impl()` is also very dependent on how many times it has already been executed.

RTS prediction

In this work, the exact number of iterations in the internal loops in `primal_net_simplex()` and `price_out_impl()` could not be found in advance as it depends on results found during the loops. This makes it difficult to predict the workload, especially for `primal_net_simplex()`. The state of the while loop could potentially be used for the prediction, but no satisfying relationship could be found from inspecting the code. Other parameters can be identified as well, which might give accurate cycle count predictions. An example is shown in Figure 5.18, where the value of a parameter `active_arcs` is shown. There might be a useful relationship between the `active_arcs` parameter and the cycle count in each while iteration, but the investigation of this is left for future work.

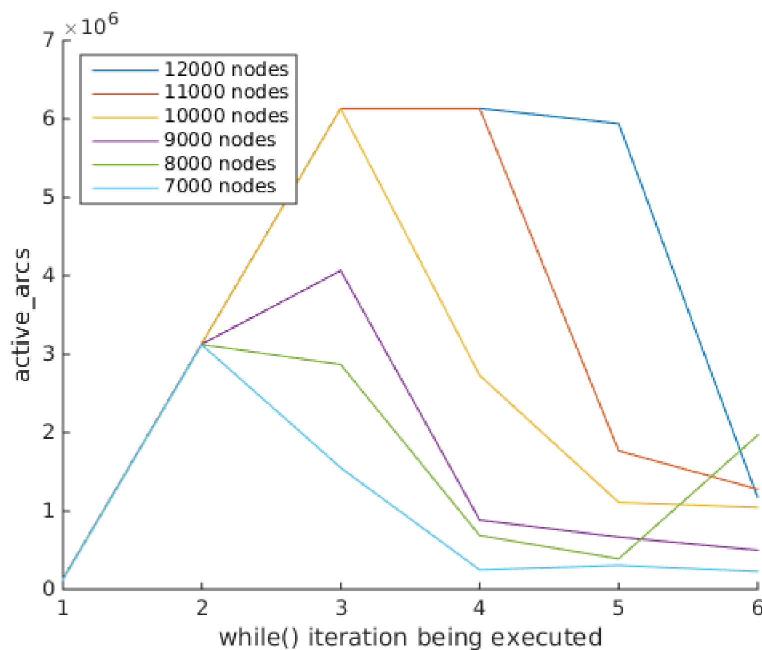


Figure 5.18: The number of `active_arcs`. This parameter is available from the beginning of every while iteration, and can potentially be used as a predictor for the cycle count in each while iteration.

From the inconsistency seen in Figure 5.16 and 5.17 it is considered infeasible to accurately calculate the number of necessary cycles within each function call. It is also observed that data sets generated with one or two extra nodes can give much variation in the cycle count of these functions. Therefore, a probabilistic approach is chosen instead of analyzing code and studying line annotated profiling. As shown in Figure 5.19, the relationship between input nodes and total CPU cycle count is quite consistent, so the entire processing of an input can be viewed as one RTS. From quadratic regression of the profiling results, an expression relating the CPU cycles to the input nodes is found, also shown in Figure 5.19. The number of input nodes is therefore used as the DVFS RTS parameter.

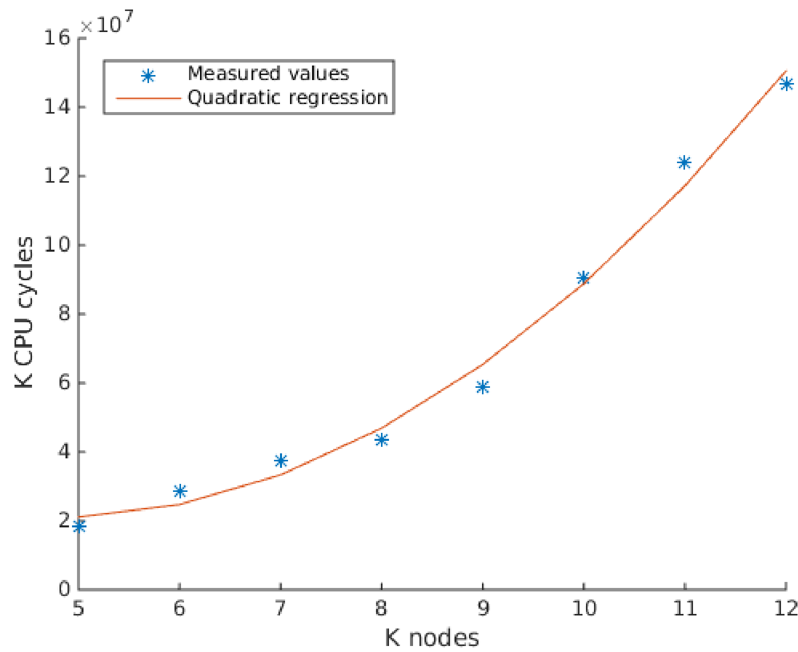


Figure 5.19: Measured CPU cycles and the result of quadratic regression.

DVFS sub-scenarios

If a continuous set of RTSs from 5000 to 12000 nodes is assumed, the regression results shown in Figure 5.19 can be used to calculate the necessary cycles for any number of input nodes. Then, the worst-case RTS that can be processed at each operating point can be calculated, as was done with the prime number checker (Section 5.1).

For this sub-scenario development however, only the actual measurements shown in Figure 5.19 are used for the DVFS sub-scenario generation. When no deadline is considered, each RTS can be executed in each of the operating points, giving one DVFS sub-scenario. When a deadline of 90 seconds is assumed (just enough for the 12,000 nodes input to finish at maximum performance), the DVFS sub-scenarios in Table 5.6 are obtained.

Table 5.6: Sub-scenario table for MCF RTSs.

Worst-case RTS parameter	Minimum operating frequency	Sub-scenario
8000	0.6GHz	1
9000	0.8GHz	2
10,000	1.2GHz	3
11,000	1.6GHz	4
Backup	1.6GHz	4

For the rest of the scenario development, RTSs with 6000, 8000, 10000 and 12000 nodes are considered in order to simplify the demonstration of the methodology.

5.3.2 Memory dynamism

RTS identification

Through profiling with Massif, as described in Subsection 4.2.2, it is observed that the main memory allocation happens in `resize_prob()`, when the arc array is resized in `price_out_impl()`. Figure 5.20 shows the allocated memory as a function of current instruction number. The size of the allocated memory is practically constant, but the time of the allocation depends on input, as can be seen in Figure 5.20.

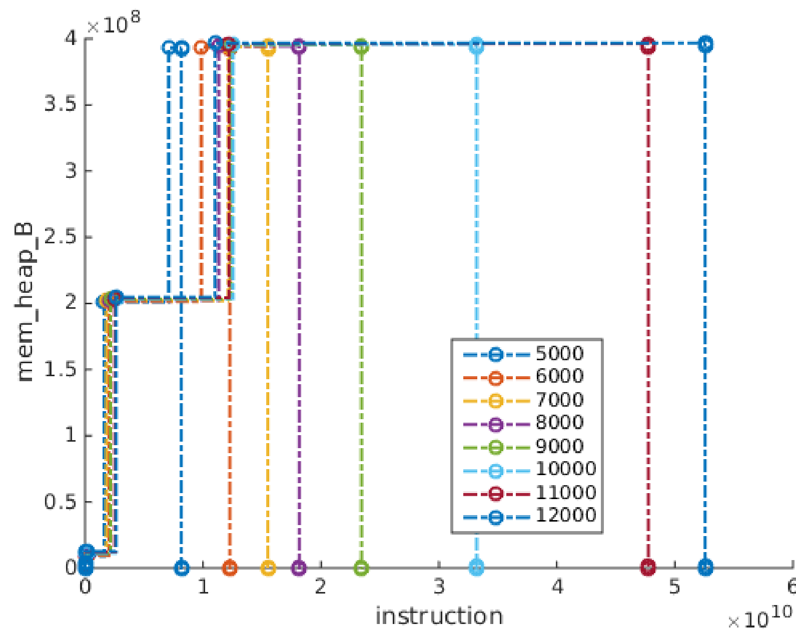


Figure 5.20: Memory allocation in MCF. Note that the allocation mainly happens in two chunks.

This dynamism in allocation time can be exploited. The two main allocations are performed from the same line in the source code; the first allocation happens in the first while iteration, and the second in the second while iteration. The size of the input affects the amount of processing that is done in other parts of the application, which is what causes the timing dynamism. Four input sizes will be tested; 6000, 8000, 10000 and 12000 nodes. Splitting the processing of each input in two RTSs results in a total of eight RTSs.

RTS prediction

As shown in Table 5.7, the number of loads and stores depends on the number of input nodes, while the maximum allocated memory is fairly independent of this. Since the necessary memory from the beginning of each execution is the same, regardless of input, there is no need for predicting this. The timing of the second allocation should however be predicted as accurately as possible in order to save memory energy. As already mentioned, the second allocation happens when the arc array is resized in `price_out_impl()`, during the second while loop iteration. When a certain point is reached in code, e.g. at the end of `primal_net_simplex()` in the second while loop, the full memory capacity can be

enabled, just in time before the allocation. The while loop iteration variable is therefore used as the memory RTS parameter.

Table 5.7: Memory use characteristics of MCF.

RTS	Input nodes	Allocation	Max. allocated mem.	Loads	Stores
1	6000	1	192MB	3507204617	740253860
2	6000	2	375MB	996936727	87538826
3	8000	1	193MB	4031558889	919078223
4	8000	2	376MB	2702998817	295775550
5	10000	1	194MB	4506000995	1005874469
6	10000	2	377MB	7941498283	1060150832
7	12000	1	195MB	3941369747	990589828
8	12000	2	378MB	15114875277	3474715047

Memory sub-scenarios

To exploit the dynamism in allocated memory size, the RTSs are grouped into two different memory sub-scenarios; 196MB and 380MB. Assuming the memory platform described in Section 3.2, the following set of memory bank configurations are suggested for the application:

a) 512MB

b) 512MB + 256MB

c) 256MB + 128MB

d) $3 \cdot 128$MB

e) 256MB + 128MB + 64MB + 4MB

With each configuration, the minimal amount of memory will be enabled for each RTS.

5.3.3 Exploration and Exploitation

Now the identified dynamism and sub-scenarios are combined and optimized using Algorithm 1 and 2. The RTS sequence for which the optimization is performed consists of three inputs with 6000, 8000, 10000 and 12000 nodes. Only these four inputs are used in order to simplify the demonstration of the methodology. The memory RTSs are more fine-grained than the DVFS RTSs, so to combine the DVFS and memory sub-scenarios, each input must be split into two RTSs (like the memory RTSs are). RTS 1 and 2 are the first and second parts of the 6000 node input, similarly RTS 3 and 4 are from the 8000 node input and so on. The estimated energy consumption for the different configurations when no deadline is assumed can be seen in Figure 5.21. There is a surprising similarity between the RTSs in the two identified memory sub-scenarios (RTS 1, 3, 5, 7 and RTS 2, 4, 6, 8). The optimal operating point and configuration do not depend on the input size, but only on whether the first or the second memory sub-scenario is active. The reason

why 1.6GHz is always optimal for the even numbered RTSs is that in these RTSs, the total memory has been enabled. Running at a high frequency reduces the memory standby energy considerably. The optimal memory configuration when all RTSs are considered is configuration c) 256MB + 128MB.

Figure 5.21 gives the following two scenarios:

- Scenario 1: 265MB bank is active, 128MB bank is shut off, CPU operating at 1.2GHz
- Scenario 2: 265MB bank is active, 128MB bank is active, CPU operating at 1.6GHz
- Backup scenario = Scenario 2

RTS 1, 3, 5 and 7 are grouped to scenario 1 and RTS 2, 4, 6 and 8 to scenario 2. Even though the RTSs within each scenario then will have different CPU cycle counts, they have the same optimal operating point and active bank combination, unless a deadline is introduced.

A deadline (and a corresponding input period) of 90 seconds is now assumed for processing each input, giving just enough time for the 12,000 node input to finish within the deadline. Figure 5.22 shows the result. Configuration c) is still optimal all RTSs considered, but almost all the optimal operation points have changed. The minimum frequency is now preferred, as it reduces the waiting time between the inputs. For RTS 5-6, the lowest frequency possible without breaking deadlines is 1.2GHz and correspondingly 1.6GHz for RTS 7-8. This means no change of operating point within each DVFS sub-scenario.

Figure 5.22 gives the following scenarios:

- Scenario 1: 265MB bank is active, 128MB bank is shut off, CPU operating at 0.6GHz
- Scenario 2: 265MB bank is active, 128MB bank is active, CPU operating at 0.6GHz
- Scenario 3: 265MB bank is active, 128MB bank is shut off, CPU operating at 1.2GHz
- Scenario 4: 265MB bank is active, 128MB bank is active, CPU operating at 1.2GHz
- Scenario 5: 265MB bank is active, 128MB bank is shut off, CPU operating at 1.6GHz
- Scenario 6: 265MB bank is active, 128MB bank is active, CPU operating at 1.6GHz
- Backup scenario = Scenario 6

5.3.4 Prediction and Switching

When no deadline is assumed, only the memory RTS parameter is used to predict the next scenario, i.e. the loop variable in the main while loop. At the beginning of each application execution, the first scenario is entered. When `primal_net_simplex()` in the second while iteration is finished, i.e. just before the second main allocation is about to happen, it is switched to scenario 2. The CPU and memory behavior seen in Figure 5.19 and 5.20 make it reasonable to believe that the identified scenarios should be applicable for any number of input nodes. The prediction and switching should be performed as follows:

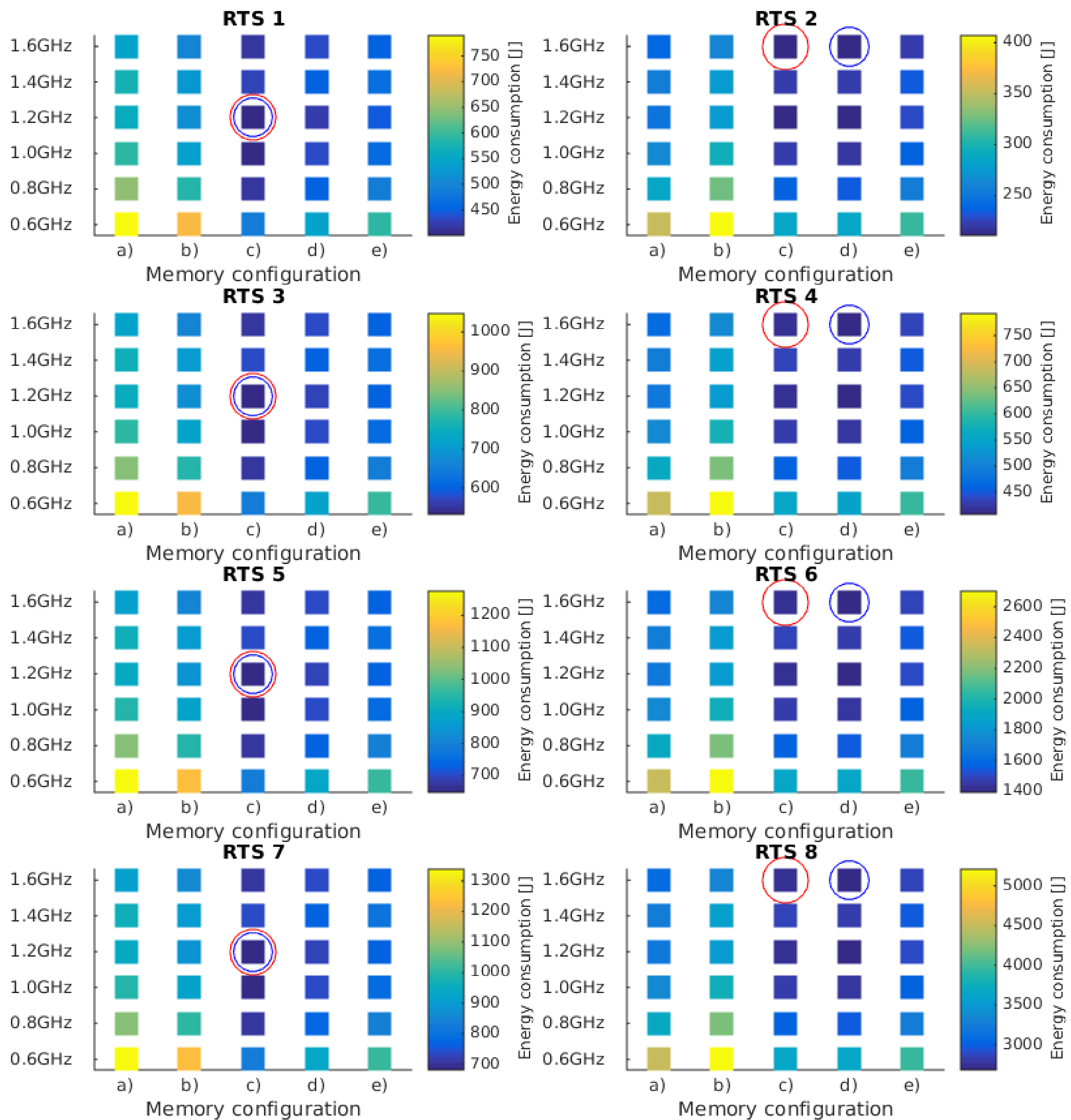


Figure 5.21: The different energy consumptions of each RTS depending on CPU operating point, denoted by frequency on x-axis. The blue circle marks the optimal combination of CPU operating point and memory configuration for each RTS individually, and the red circle marks the best CPU operating point for each RTS given the best memory configuration for all the RTSs combined.

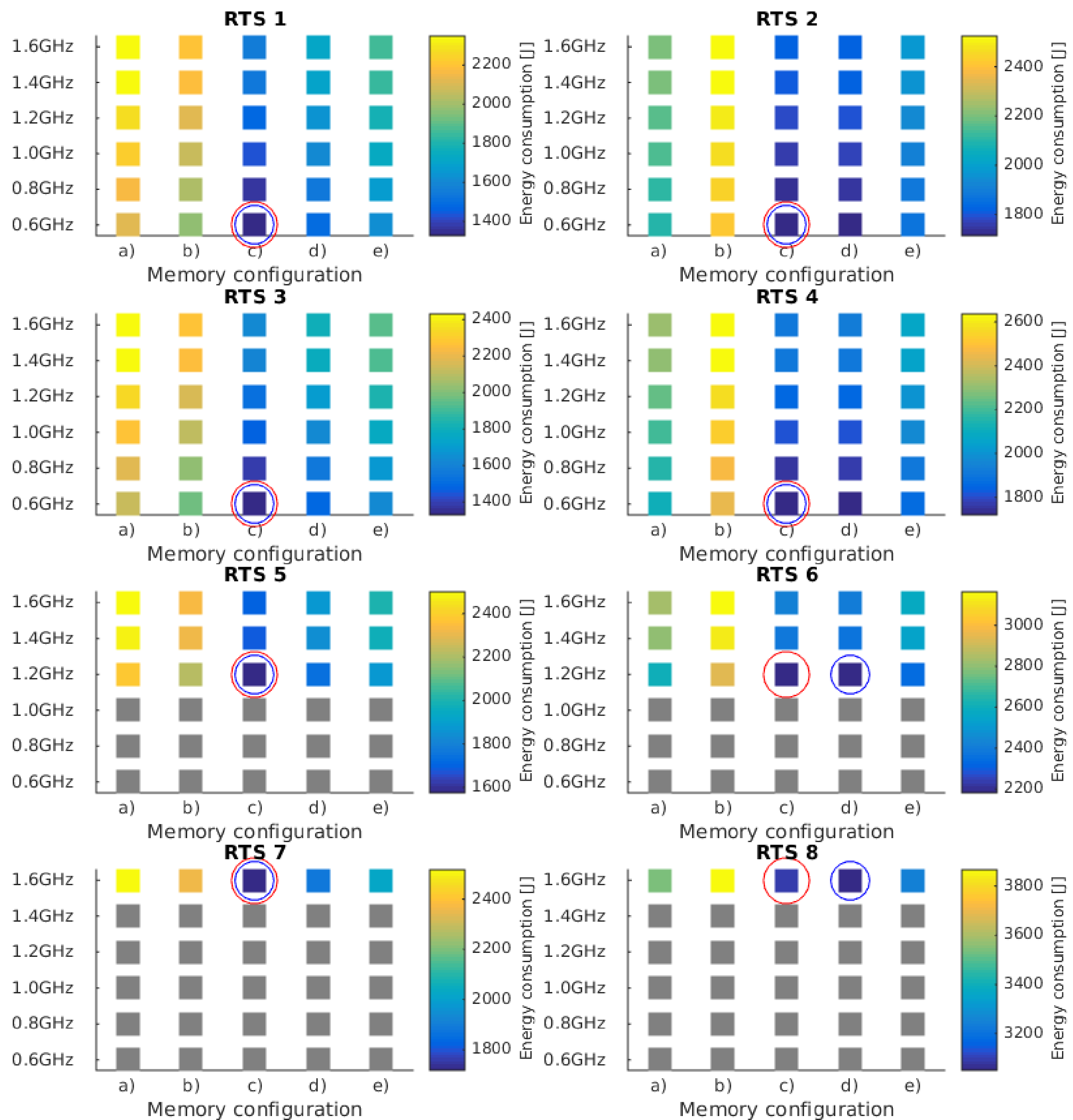


Figure 5.22: With deadline of 90 seconds. Blue circle marks the optimal combination of CPU operating point and memory configuration for each RTS individually, and red circle marks the best CPU operating point for each RTS given the best memory configuration for all the RTSs combined.

- From the beginning of each input processing, if the number of input nodes is between 5000 and 12000, use scenario 1.
- When `primal_net_simplex()` finishes in the second while iteration, if the number of input nodes is between 5000 and 12000, use scenario 2.
- Otherwise, use the backup scenario.

When the deadline is assumed, the number of input nodes must be used as an RTS parameter in addition to the while loop variable. The same operating point is used for both RTSs in each DVFS sub-scenario, i.e. constant operating point for each input. This should be set as soon as the number of input nodes is known. From the beginning of each input processing;

- If the number of input nodes is up to 8,000, use scenario 1.
- If the number of input nodes is up to 10,000, use scenario 3.
- If the number of input nodes is up to 12,000, use scenario 5.

As without deadline, when `primal_net_simplex()` in the second while iteration is finished, the second switch should happen;

- If the number of input nodes is up to 8,000, use scenario 2.
- If the number of input nodes is up to 10,000, use scenario 4.
- If the number of input nodes is up to 12,000, use scenario 6.

The memory behavior seen in Figure 5.19 and 5.20 still gives reason to believe that the memory sub-scenarios are valid for any number of input nodes between 5,000 and 12,000. The identified scenarios can be extended to any number of input nodes by using the regression shown in Figure 5.19. The lowest possible frequency for an RTS should be optimal, so the worst-case RTS parameter (number of input nodes) for each operating point can be calculated at design-time and put in a lookup table. At run-time, the current number of input nodes can be compared to the entries in the lookup table to find the operating point with the lowest possible frequency.

A mechanism like the scenario manager described in Section 3.3 can be implemented. When the deadline is assumed we need two different prediction mechanisms however, distributed to the two different locations in the source code where each scenario should be entered. These mechanisms will call the switching mechanism to configure the system according to the predicted scenario. The CPU unavailability time of 10μs per change of operating point can be neglected when compared to the average execution time of each scenario. So can the additional CPU cycles of the scenario management. A memory bank is awakened only once during the processing of each input, so the wake up energy for the memory can also be neglected.

5.3.5 Results

Figure 5.23 and 5.24 show the total energy consumption for the different memory configurations without and with deadline respectively. The leftmost point in both plots is a static implementation using the highest CPU frequency and a 512MB memory bank regardless of RTS. Without deadline, the total energy reduction when comparing the most optimal implementation to the static implementation is 16.9%. With deadline, the energy reduction is 29.1%. Figure 5.25 and 5.26 compares the static implementation to the scenario implementation for each RTS separately, without and with deadline respectively. From these figures it can be seen that the CPU energy dominates without deadline, and the memory energy dominates with deadline. This is because the memory standby energy becomes substantial with the increased idle time between inputs. Considering the deadline version, it is clear that the smaller input sizes benefit the most from the scenario implementation. Especially RTS 1 and 3, where the energy is reduced with 43% and 45% respectively. An increased occurrence of these RTSs would thus make the scenario implementation even more beneficial.

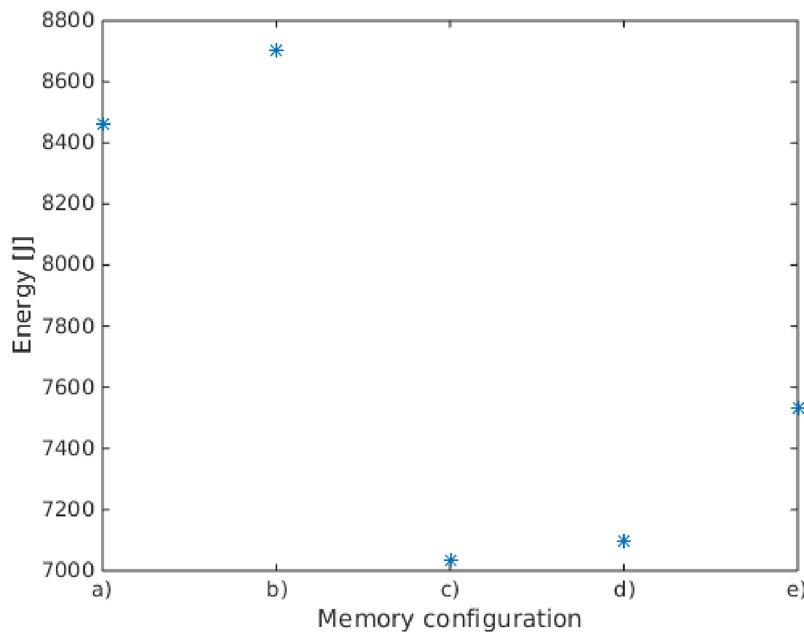


Figure 5.23: Total energy of all the RTSs for each of the different memory configurations. Each RTS is executed in its optimal operating point. No deadline.

5.3.6 Other observations

It is also noted that the average Cycles Per Instruction (CPI) for this application is very high, due to the many cache misses [31]. This is a result of intensive network computation and manipulation with severe pointer chasing. Faster and more energy-efficient memory accesses therefore have big potential to reduce energy, as it would mean less waiting time for the CPU. By partitioning the memory into several memory banks, the energy per

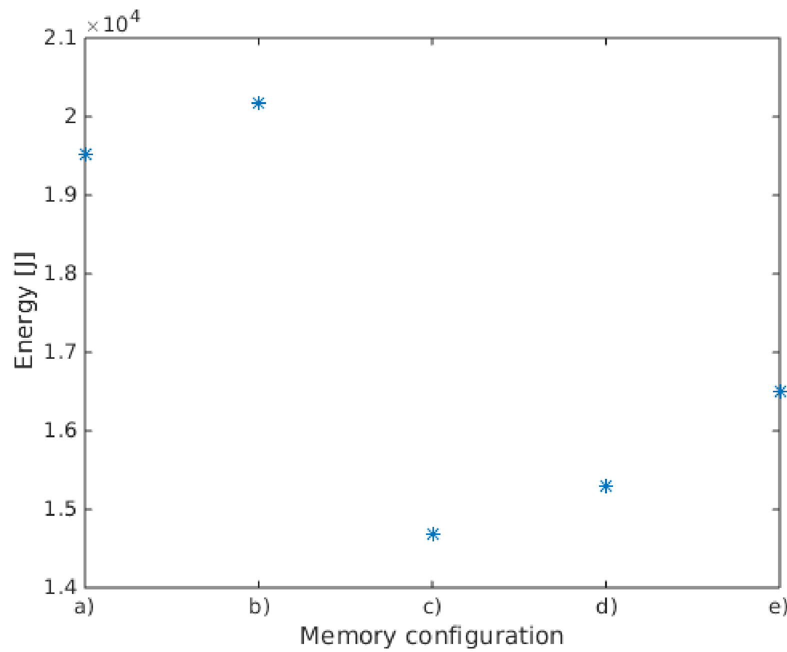


Figure 5.24: Total energy of all the RTSs for each of the different memory configurations. Each RTS is executed in its optimal operating point. Deadline of 90 seconds is assumed.

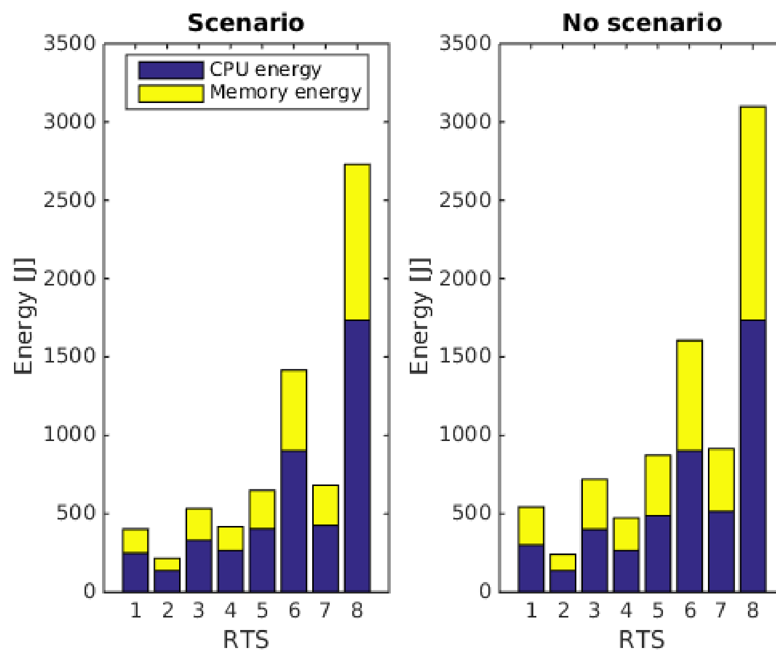


Figure 5.25: Comparison of the total energy consumption of the static, no scenario implementation versus the optimal system scenario implementation for each RTS. No deadline.

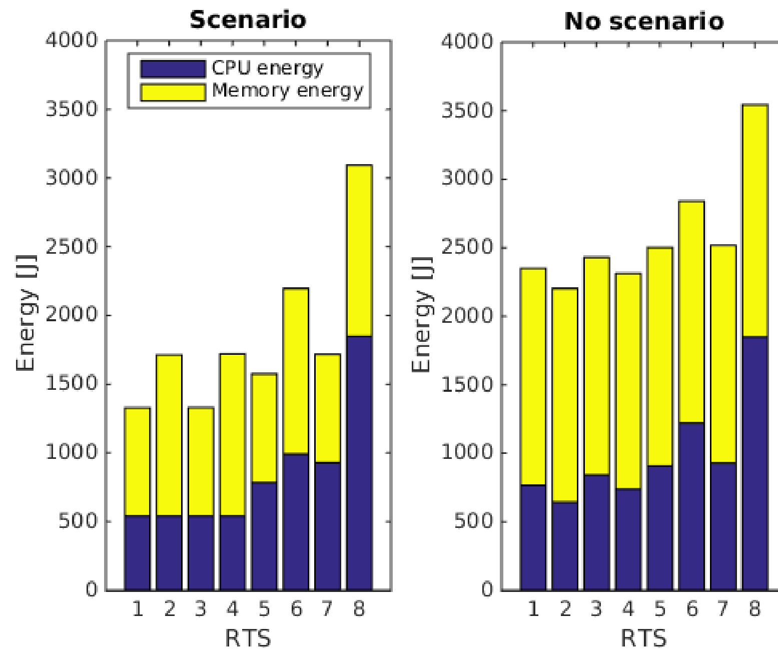


Figure 5.26: Comparison of the total energy consumption of the static, no scenario implementation versus the optimal system scenario implementation for each RTS. Deadline of 90 seconds is assumed.

access can be reduced, and more accesses can happen in parallel. The CPUs operating point could also be adapted to match memory access speed using DVFS.

5.4 464.h264ref

This benchmark is a reference implementation of the encoder used in H.264/AVC, a state-of-the-art video compression standard. The standard is widely used, especially within high-resolution applications such as Blu-ray and HD video broadcasting [16]. Two video streams are encoded in the benchmark; "foreman_qcif.yuv" and "sss.yuv". *Foreman* is a much used video stream in video compression research, here with 120 frames and 176x144 pixel resolution. The other sequence, *sss*, is a sequence of 171 frames, 512x320 pixels, taken from a video game.

The only actual input to the H.264 encoder benchmark is a config-file that specifies the path to the video stream and the system parameters to use. Two system parameter profiles are used in the benchmark; *baseline* and *main*. The baseline profile provides good compression and fast encoding, and is typically used in applications that require real-time encoding (e.g. video conference). The main profile gives a better compression and is used when no loss of data can occur. There is also a wide range of other settings that can be made, such as frame rate, width and height, and more advanced settings that e.g. specifies how the motion estimation should be performed.

Five different types of frames are supported by the H.264/AVC standard: I, P, B, SP and SI frames. I frames are *intracoded*, i.e. only blocks within the same frame are compared during the encoding. P and B frames are *intercoded*. P frames are coded by comparing with previous I and P frames (forward prediction), and B frames are compared both to previous and future frames (bidirectional prediction). SP and SI frames are specially encoded frames, e.g., used in transitions from one bit rate to another, frame skipping, fast forward, and more [49]. These frame types are not used in the benchmark suite. The baseline encoding profile only uses I and P frames, which is why it is the preferred profile for real-time encoding like in video conferences. The main encoding profile uses both I, P and B frames, and can involve complex frame reordering because of this. Below is an example that shows how the different frame types (I, P, B) are ordered with baseline and main encoding, taken from the benchmark output. The numbers behind each frame are the original frame numbers from the sequence.

- Baseline: I (0), P (1), P (2), P (3), P (4), P (5), P (6)
- Main: I (0), P (2), B (1), P (4), B (3), I (6), B (5)

The resource requirements of encoding a frame depend on the frame type. This frame-level dynamism can be exploited with system scenarios, similarly to what is done with the MPEG-2 encoder in [17]. Yassin et al. [39] uses dynamism caused by varying frame sizes to implement DFS system scenarios with the H.264 encoder control structure. The encoding profile should also affect the execution time, according to the benchmark description [16]. These kinds of dynamism will be investigated in this work, except frame-level dynamism which is left for future work.

5.4.1 DVFS dynamism

RTS identification

There are many parameters to modify in this benchmark, both regarding the input stream properties and the applied encoding technique. This subsection investigates the CPU

cycle count dynamism due to frame size, content, length of the sequence and the type of encoding profile that is used. The input set is limited to the video streams included in the benchmark. Frame size and sequence length is adjusted through the config-file.

Through profiling with Zoom, it is observed that different sequences with equal dimensions and frame count that are encoded with the same profile have very similar CPU cycle count, i.e. the content of the stream does not cause dynamism in the tested case. By keeping all parameters constant except the input sequence length, it is observed that an increased sequence length gives a proportionally increased number of CPU cycles. The CPU cycle count is also quite proportional to the product of the height and width of the encoded frames. Furthermore, main profile encoding uses about twice as many CPU cycles as baseline encoding, when the other parameters are kept constant. Figure 5.28 shows the dynamic CPU cycle count as a function of pixels per input frame for baseline and main profile encoding.

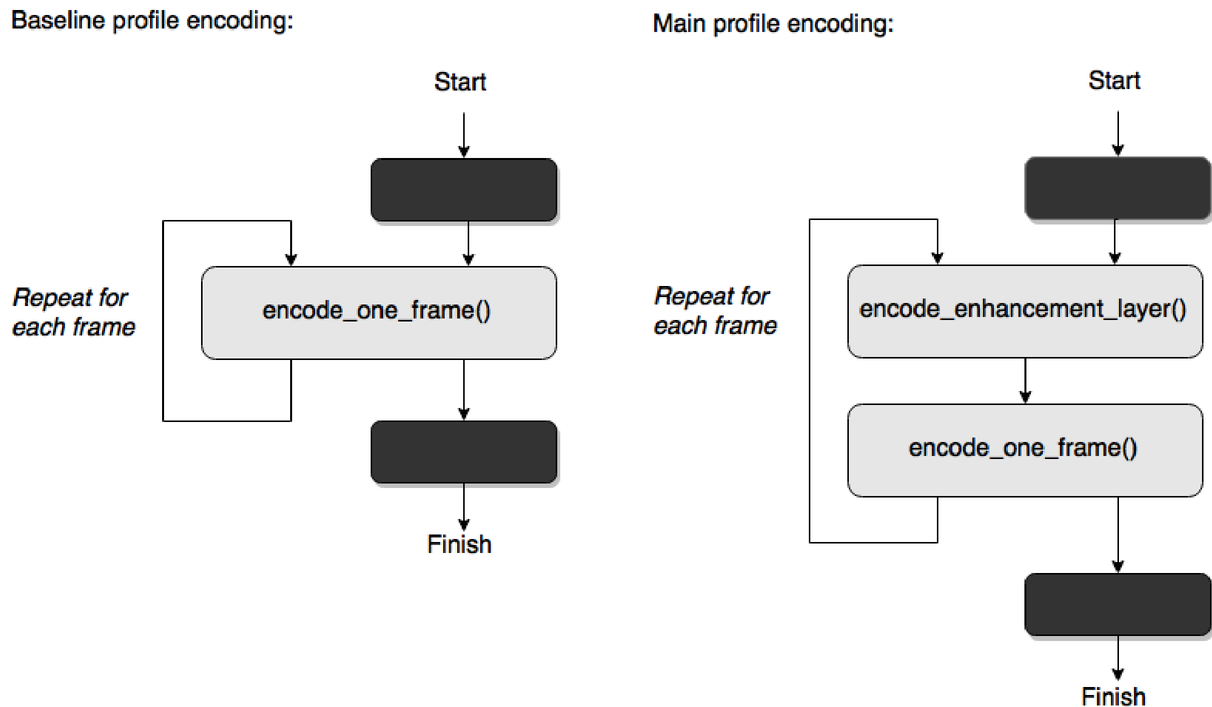


Figure 5.27: H264 top-level model.

The application is decomposed according to the procedure in Subsection 4.2.1 as shown in Figure 5.27. In Zoom it can be seen that the function `encode_one_frame()` is called from the main routine once for each frame, both with baseline and main profile encoding. The function `encode_enhancement_layer()` is also called from the main routine once for every frame, but only executes considerable code for B-frames (bi-directionally predicted frame), i.e. when main encoding is used. Most of its cycles are actually also from calling `encode_one_frame()`. All together, 99% of the CPU cycles are thus spent in `encode_one_frame()` (and the functions called by this function) both for main and baseline encoding. The rest of the function call hierarchy is complex and deep, so no further investigation of this is presented.

The benchmark is profiled while encoding scaled versions of ”sss.yuv” to study the relationship between encoding profiles and frame sizes. The frame sizes are shown in Table 5.8, and the profiling results using sequences of 15 frames are shown in Figure 5.28. For the rest of the system scenario development, the considered input set comprises streams of the first four of these frame sizes, with 15 frames of each stream. This can be a realistic situation for encoders that are capable of some frame buffering. Both baseline and main encoding of each stream is considered, producing a total of eight RTSs.

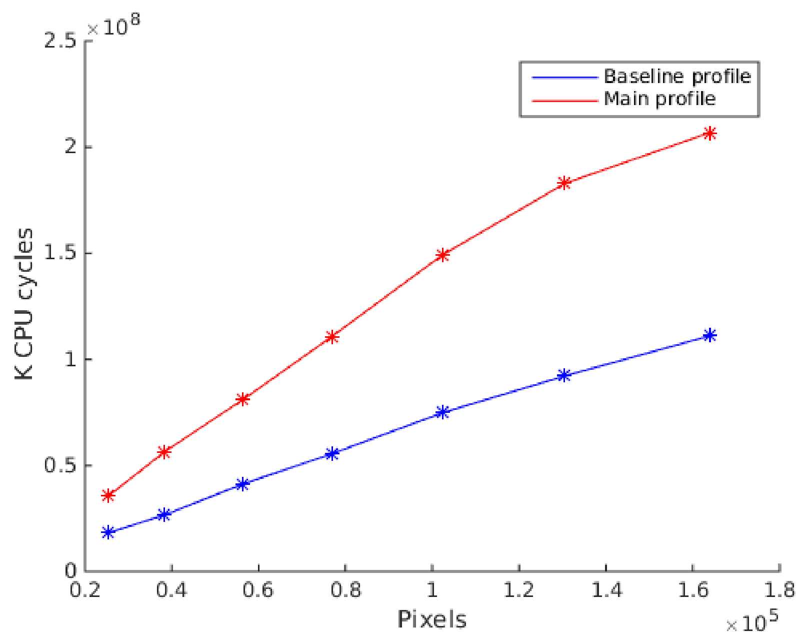


Figure 5.28: CPU cycles as a function of the number of pixels per frame for a 15 frame sequence.

Table 5.8: ”sss.yuv” is scaled to the following frame sizes.

Frame size	Pixels
176x144	25344
240x160	38400
320x176	56320
320x240	76800
320x320	102400
480x272	130560
512x320	163840

RTS prediction

In streaming applications, the encoding deadline usually depends on the frame rate. The RTSs should then be predicted from frame size, profile and frame rate, so that the frames

are encoded at the same rate that new frames are arriving. The same goes for non-streaming applications, except that the total number of frames can be used instead of the frame rate.

The RTSs considered here can be predicted from frame size and encoding profile. Both are known at the beginning of each encoding as these properties are specified in the config-file. Only the first four frame sizes are used as input for the remaining parts of this section.

DVFS sub-scenarios

If no deadlines must be met, all RTSs can be executed in all operating points. When the deadline is set to 70 seconds, which gives just enough time for the worst-case RTS to finish using the highest performance operating point, the sub-scenario table shown in Table 5.9 is obtained.

Table 5.9: DVFS sub-scenarios.

Input	Encoding profile	RTS	Min. operating freq.	Sub-scenario
sss_176x144	baseline	1	0.6GHz	1
sss_240x160	baseline	2	0.6GHz	1
sss_320x176	baseline	3	0.6GHz	1
sss_320x240	baseline	4	0.8GHz	2
sss_176x144	main	5	0.6GHz	1
sss_240x160	main	6	1.0GHz	3
sss_320x176	main	7	1.2GHz	4
sss_320x240	main	8	1.6GHz	5
Anything else			1.6GHz	Backup

5.4.2 Memory dynamism

RTS identification

The two different encoding profiles have quite different memory footprint. Encoding the same sequence with the baseline profile requires about 50% more memory than encoding with the main profile. As noted earlier, main profile encoding requires significantly more cycles than baseline profile encoding. Figure 5.29 and 5.30 show the profiling results from encoding different inputs using the baseline and main profile settings. From these figures it is concluded that the dynamism in memory allocation timing will be difficult to exploit, since much of the allocations happen relatively early. The dynamism in memory allocation size can however be exploited with memory-aware system scenarios. The memory power consumption can potentially be reduced a lot for the low resolution input sequences, especially when the main profile is used. The eight RTSs listed in Table 5.10 are investigated.

RTS prediction

The same two RTS parameters that were used with the CPU RTSs are used for the memory RTSs as well, i.e. the frame size and the encoding profile. Both of these are

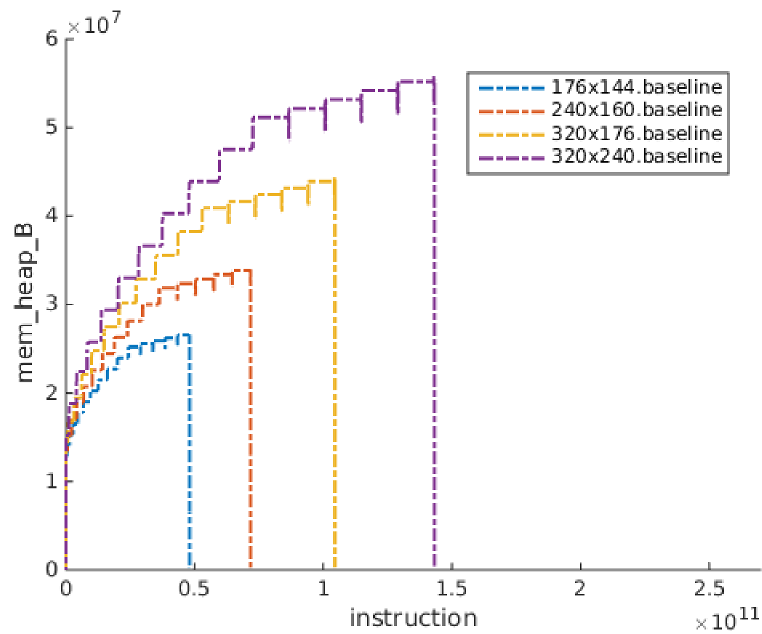


Figure 5.29: Memory allocation in H.264 with baseline profile encoding.

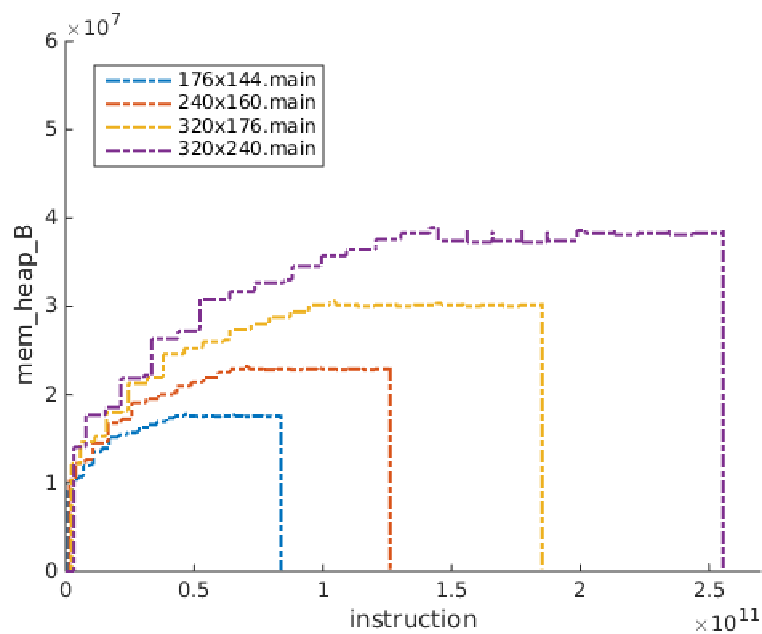


Figure 5.30: Memory allocation in H.264 with main profile encoding.

Table 5.10: The maximum memory allocated by the H.264 encoder when processing 15 frames of *sss* with various frame sizes.

Frame size	Baseline	Main
176x144	25.5MB (RTS 1)	17.0MB (RTS 5)
240x160	32.5MB (RTS 2)	22.2MB (RTS 6)
320x176	42.3MB (RTS 3)	29.2MB (RTS 7)
320x240	53.3MB (RTS 4)	37.2MB (RTS 8)

known at the start of the execution as they are specified explicitly in the config-file.

Memory sub-scenarios

From the identified memory dynamism shown in Table 5.10 and the available memory bank sizes presented in Section 3.2, the following seven memory bank configurations are suggested:

a) 32MB · 2 (static)

b) 32MB · 2 (dynamic)

c) 32MB + 16MB · 2

d) 32MB + 16MB + 8MB

e) 32MB + 16MB + 8MB + 4MB

f) 32MB + 16MB + 8MB + 4MB + 2MB

g) 32MB + 8MB · 2 + 4MB + 2MB

With each configuration, the minimal amount of memory will be enabled for each RTS. This gives the memory sub-scenarios.

5.4.3 Exploration and Exploitation

The different combinations of DVFS and memory sub-scenarios are explored and exploited using Algorithm 1 and 2. The result when no deadline is assumed is shown in Figure 5.31, where it can be seen that the only optimal operation point is at 1.6GHz, regardless of input. This is because of the big memory requirements relative to the execution time, as will be shown in the next subsection. By running at the highest frequency available, the execution time is shortened and less standby energy is consumed by the memories. Only one DVFS sub-scenario is therefore used; 1.6GHz operation. The memory sub-scenarios are however used to switch memory banks on and off depending on the input. The resulting scenarios are shown in Table 5.11.

Giving a deadline does by itself not change the results, as the fastest execution is already optimal. Quite different results are however obtained when assuming that new inputs are received on each deadline and the CPU is left idle while waiting for the next input. Figure 5.32 shows the result when the deadline is set to 70 seconds, which is

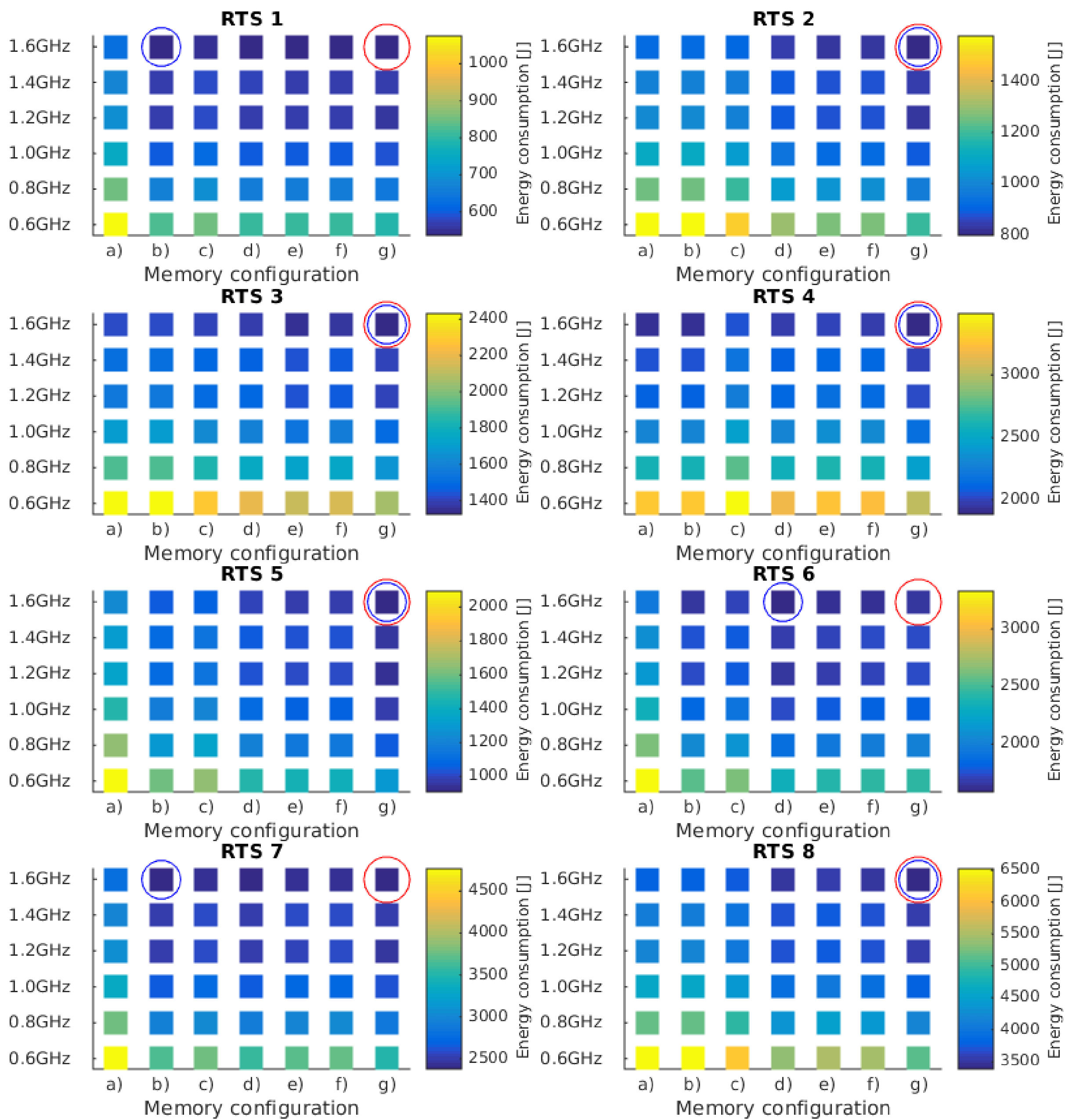


Figure 5.31: RTS platform exploration for the RTSs given in Table 5.10. No deadline.

Table 5.11: The H.264 scenarios when no deadline is assumed. The different active bank combinations for the RTSs (i.e. memory sub-scenarios) defines the scenarios. The optimal CPU operating point is 1.6GHz for all the RTSs.

Input, encoding profile	**RTS**	**Active memory banks**	**Scenario**
sss_176x144, baseline	1	32MB	1
sss_240x160, baseline	2	32MB + 2MB	2
sss_320x176, baseline	3	32MB + 8MB + 4MB	3
sss_320x240, baseline	4	32MB + 2 · 8MB + 4MB + 2MB	4
sss_176x144, main	5	2 · 8MB + 2MB	5
sss_240x160, main	6	32MB	1
sss_320x176, main	7	32MB	1
sss_320x240, main	8	32MB + 4MB + 2MB	6
Anything else	-	32MB + 2 · 8MB + 4MB + 2MB	Backup

just enough for the worst-case RTS (RTS 8) to finish within the deadline. The same memory configuration is still optimal, so the same active bank combinations are used. The optimal CPU operating points have changed however. The result is now one scenario for each RTS, as shown in Table 5.12. Some of the scenarios can be merged without much penalty. For example scenario 6 can be merged into scenario 7, as they have the same active bank combination and Figure 5.32 shows similar energy consumption in operating point 1.0GHZ and 1.2GHz for RTS 6.

Table 5.12: H.264 scenarios when a deadline is assumed. The different active bank combination and optimal CPU operating point (denoted by frequency) for each RTS results in a scenario for each RTS.

Input, encoding	**RTS**	**CPU op**	**Active memory banks**	**Scenario**
sss_176x144, baseline	1	0.6GHz	32MB	1
sss_240x160, baseline	2	0.6GHz	32MB + 2MB	2
sss_320x176, baseline	3	0.6GHz	32MB + 8MB + 4MB	3
sss_320x240, baseline	4	0.8GHz	32MB + 2 · 8MB + 4MB + 2MB	4
sss_176x144, main	5	0.6GHz	2 · 8MB + 2MB	5
sss_240x160, main	6	1.0GHz	32MB	6
sss_320x176, main	7	1.2GHz	32MB	7
sss_320x240, main	8	1.6GHz	32MB + 4MB + 2MB	8
Anything else	-	1.6GHz	32MB + 2 · 8MB + 4MB + 2MB	Backup

5.4.4 Prediction and Switching

The same two RTS parameters have been identified for the CPU and memory dynamism; the frame size and the encoding profile. Upon receiving a new input sequence to encode, these parameters can be extracted from the config-file. Then the optimal scenario and its configurations are found from a lookup table (like Table 5.11 or 5.12). A mechanism like the scenario manager described in Section 3.3 should be implemented to configure

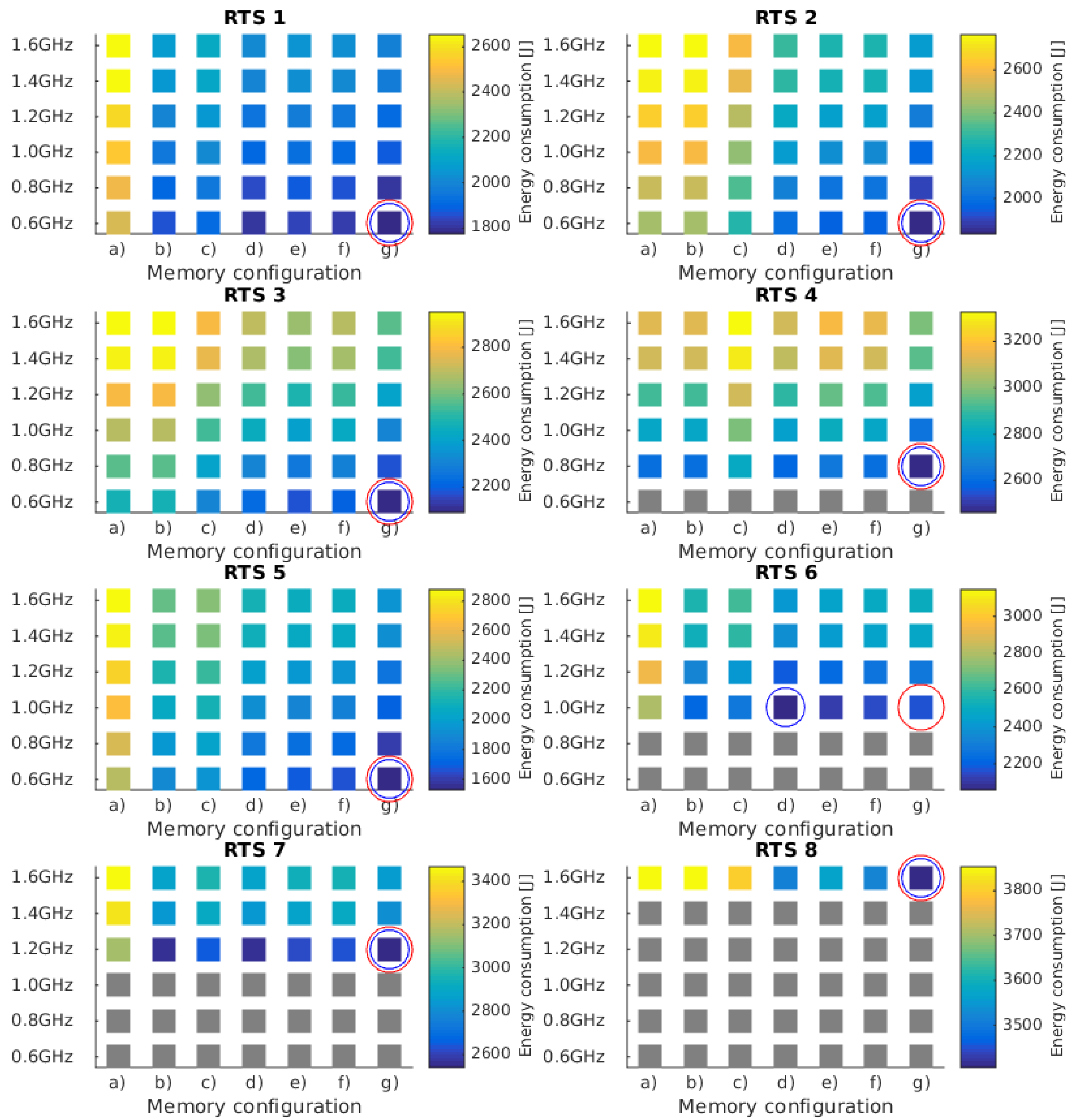


Figure 5.32: RTS platform exploration for the RTSs given in Table 5.10 when the deadline for encoding each input is set to 70 seconds and the CPU is left idle between each input.

the system according to the scenario. The scenario switch should happen before any of the input processing is started.

Also for this application, the CPU and memory overhead for scenario prediction and switching can be neglected.

5.4.5 Results

Figure 5.33 and 5.34 shows the estimated total energy consumption of the different memory configurations without and with deadline respectively. The leftmost point on each plot is a static implementation using the highest CPU frequency and two 32MB memory bank regardless of RTS. Without deadline, the total energy reduction when comparing the most optimal implementation to the static implementation is only 12.4%. With the 70 second deadline however, 28.4% of the total energy is saved. Figure 5.35 and 5.36 compares the energy consumption of the static implementation to the best scenario implementation without and with deadline respectively, with the contribution from CPU and memory separated for each RTS. When no deadline is assumed, the energy consumption is quite similar. The CPU energy is in fact exactly the same, since in both cases the 1.6GHz operating point is used. The whole energy reduction is due to the memory sub-scenarios, by making sure that no more memory than necessary is enabled for each RTS. When the deadline is set, the memory energy is drastically increased for the RTSs that finish before the deadline, as is observed with the other benchmarks. There reduced energy of the scenario implementation is both due to the memory and DVFS sub-scenarios. The energy reduction for RTS 5 in particular is substantial (41%), mostly due to reduced memory energy. The required memory in this RTS is only 17MB (see Table 5.10), making it the RTS with the smallest memory requirement. It is therefore very beneficial for this RTS to use memory configuration g), so that $2 \cdot 8\text{MB} + 2\text{MB} = 18\text{MB}$ can be active while the rest of the banks are shut down. As with the other benchmarks, the RTSs with small workloads have the biggest relative energy reductions when employing scenarios.

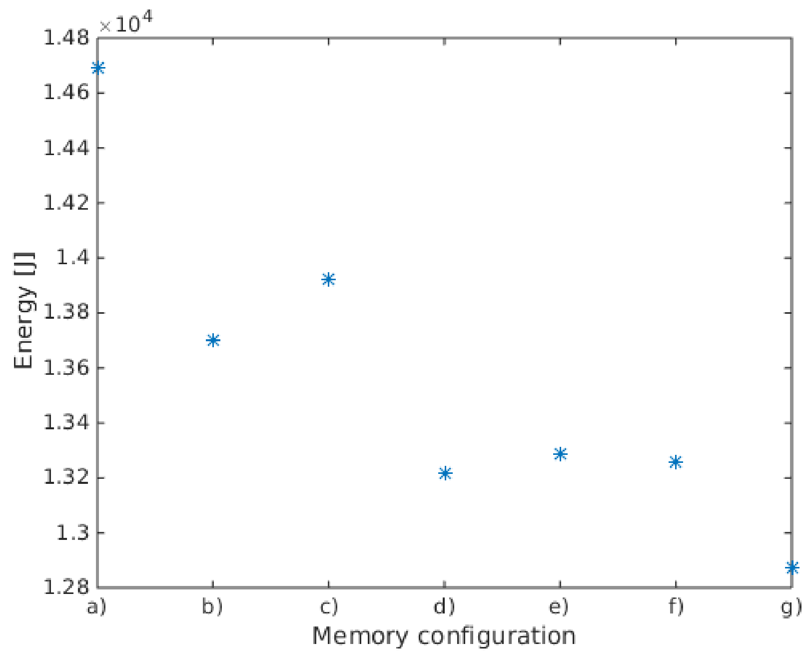


Figure 5.33: Total energy of all the RTSs for each of the different memory configurations, without deadline. Each RTS is executed in its optimal operating point.

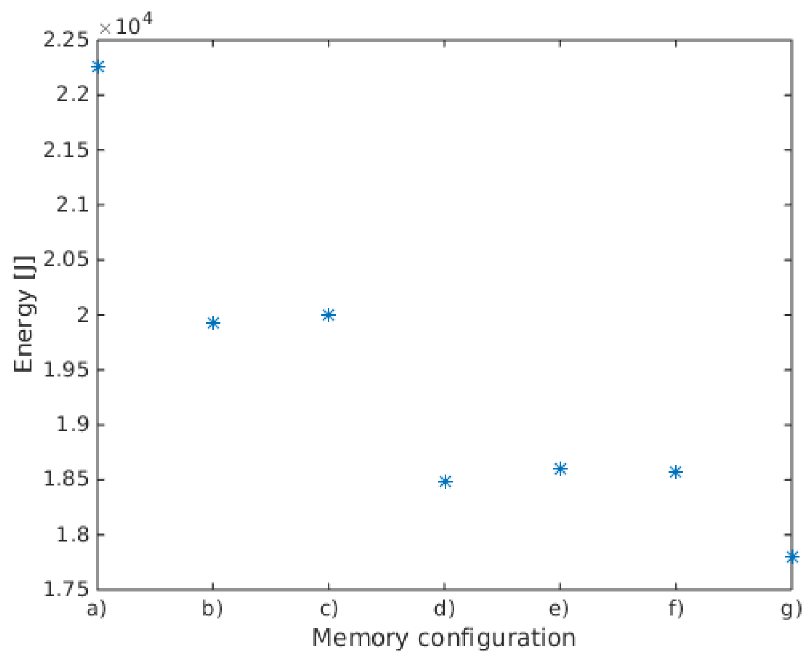


Figure 5.34: Total energy of all the RTSs for each of the different memory configurations, with deadline. Each RTS is executed in its optimal operating point.

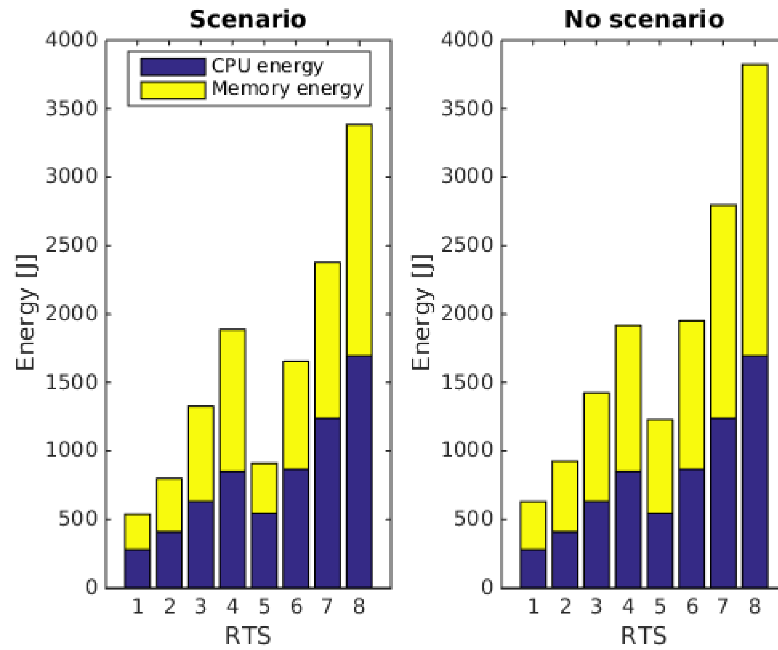


Figure 5.35: Comparison of the total energy consumption of the static, no scenario implementation versus the optimal system scenario implementation for each RTS. Without deadline.

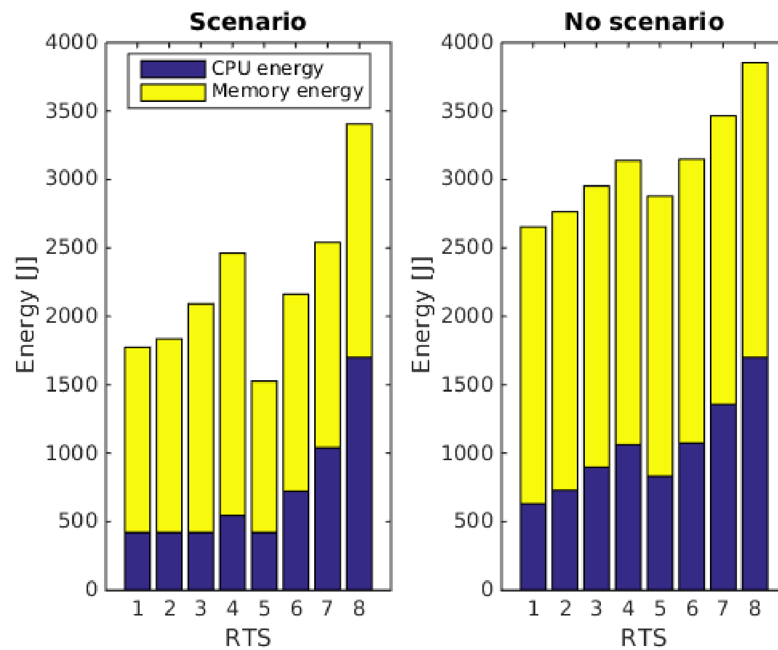


Figure 5.36: Comparison of the total energy consumption of the static, no scenario implementation versus the optimal system scenario implementation for each RTS. With deadline.

Chapter 6

Discussion and Future Work

6.1 Applications

There are huge variations between the considered applications, both when it comes to average and worst-case resource requirements. The differences in required CPU cycles and memory size results in very different energy distributions between memory and processing for the assumed platform. This is a challenge when trying to develop a unified methodology. If either the CPU or memory energy consumption makes up a very small part of the total energy consumption, the potential gain from using the presented methodology diminishes. The results of the methodology should still be optimal, but much work can be saved by only focusing on the biggest energy consumer, which can have a much bigger potential for reducing the energy. The methodology is best suited for systems where the energy consumption is more evenly distributed. As mentioned in Section 2.4, the memory energy contribution typically varies from 35% to 65% of the total system energy consumption for different architectures. Such energy distributions matches the results presented in this work (except for the prime number checker).

Around 10 to 30% of CPU and memory energy is saved by using the methodology on the selected benchmark applications. The inputs with little workload saves the most energy with system scenarios. Unfortunately, it is the more demanding inputs, in which the energy is typically not as much reduced, that have the most influence on the total energy reductions. In this work, the occurrences of the most and the least demanding inputs are equal. Dynamic applications where the most demanding inputs are rare enough to not dominate the expected total energy consumption will in general have a bigger potential for energy reductions through system scenario development. An example of this is the activity pattern of servers, as mentioned in Section 2.6.

The benchmark applications have long execution times (usually several minutes for the reference inputs on a regular workstation). The long execution times have been a challenge when designing the system scenarios, which generally required much profiling of the applications. Especially the memory profiling performed with the simulation-based profiler Cachegrind is time-consuming, as it slows down the execution time between 20 and 100 times. This has limited the profiling somewhat, resulting in fewer RTSs. Identifying and characterizing RTSs in terms of both memory and CPU utilization is a time demanding task which should clearly be automated to a much greater extent. There is a number of static profiling techniques available in literature, but dynamic techniques

are limited. A survey is presented in [17, Ch. 9], and a proposed source code profiling technique is also summarized.

The system scenario implementations introduce very little overhead in the applications relative to the application workload. Especially for the benchmark applications, where the identified RTS are all quite CPU and memory intensive (requiring somewhere between 8,943,000,000 and 814,576,000,000 CPU cycles, and between 1.4 and 377MB of memory). Even with the prime number application, which has significantly lower average CPU cycle count per RTS and very limited memory requirements, the overhead was estimated to only 0.43% additional CPU cycles. For the benchmark applications, the relative overhead should be even less due to the infrequent scenario changes.

The presented system scenario designs cannot be regarded as precise trade-off analyses for the applications, as the design decisions are founded on a theoretical platform with several simplifying assumptions. The results can however be used to reduce the search space and guide the system scenario design process for these applications. The HMMER benchmark is suited for the gray-box modelling, where the application is partitioned in order to identify and locate the dynamism. The observed execution traces are mostly independent of input, and the ending conditions of the for-loops do not require additional run-time processing to be found. The MCF benchmark on the other hand is only complicated by such a partitioning. The total cycle count is easier to predict than the cycle count from separate function calls. Also for the H.264 benchmark the total cycle count is used to predict scenarios, but a more precise cycle count prediction can be performed with an expression identified from the total loop structure, as is presented by Yassin et al. [39]. Whether the scenarios of an application should be predicted statistically or from a structural expression can be decided using application knowledge and by trial and error, as is demonstrated with the considered applications.

6.2 Memory related

All calculations are based on numbers from profiling. Ideally, actual measurements should have been made, at least to verify the models that are used. The complexity of the designed platform made it necessary to do several simplifying assumptions in the calculations. There will for example be other energy consumers than just the CPU and memory, like buses, muxes, I/O and more, that could have some impact on the results. The extra logic and wires necessary for the described memory partitioning has not been included in the calculations. This should be investigated and incorporated into the memory energy numbers from CACTI in order to get more realistic energy numbers. In many cases, replacing one bank with two smaller banks can give reduced read, write and standby energy with the memory models that are used in this work. For example, here are the characteristics (from Table 3.2) of one 8MB bank compared to two 4MB banks:

- 8MB cache:

$$P_{active} = 3.905\text{W}$$

$$P_{shutDown} = 1.628\text{W}$$

$$E_{read} = 2.188\text{nJ}$$

- $2 \cdot$ 4MB cache:

$$2 \cdot P_{active} = 2 \cdot 1.905\text{W} = 3.810\text{W}$$
$$2 \cdot P_{shutDown} = 2 \cdot 0.794\text{W} = 1.588\text{W}$$
$$2 \cdot E_{read} = 2 \cdot 0.815\text{nJ} = 1.630\text{nJ}$$

When no penalty is added for increasing the number of banks, the resulting optimal configuration might be a configuration with unrealistically many banks. For two of the applications (MCF and HMMER) studied in this work, the optimal memory configurations are however not the configurations with the most banks. The main reason for this would be that the memory banks draw power even when shut off. Furthermore, memory configurations with more than five memory banks were not explored in this work in order to limit the additional interconnects required.

The memory architectures used in this work are very simplified. Actual memory architectures would normally not be organized like the suggested configurations. For example, there is no memory hierarchy present in any of these configurations. Smarter mapping of data to memories in the context of system scenarios should be investigated in future work. In this work it has been assumed that the execution time is independent of memory configuration. For some of the considered memory banks this is far from the reality, as the variations in the time needed for a memory read can be quite big; from around 1 to 30ns according to the CACTI models, with the CPU cycle time being only 1ns at 1GHz. It is unknown how much this will affect the results, and how much techniques like pipelining and parallel data accesses can alleviate the differences. Frequently used data should anyway be mapped to the smaller banks of the memory configuration, as described in Section 2.4. Much further work can be done here.

In this work, the memory dynamism is only exploited in terms of timing and size of memory allocations. In [12] however, the memory access pattern and data reuse size is also exploited, which makes it possible to use special data assignment strategies and design memory banks specifically according to these requirements. Much energy can then be saved by organizing the data assignments according to the expected data use and by allowing memory banks to sleep when not used. If the memory accesses of the applications considered in this work are profiled in more detail, the memory sub-scenario development can be expanded to fully exploit the memory dynamism. The observed pattern in allocated memory and reads and writes indicates that there could be exploitable patterns also in the *use* of the allocated memory. If for example some memory locations are only used by a specific function, an RTS could be constructed from this. If these memory locations can be gathered in one memory bank, this bank can be kept in a low-power mode whenever the function is not executing. The memory use in HMMER and H.264 is assumed to be very sequential judging by their low cache miss rates [31] and the nature of the applications, making these applications suited for such exploitation. Alternatively, Section A.1 proposes an implementation that can also be considered for these applications. MCF on the other hand is probably not very suited, since it has a very high cache miss rate due to severe pointer chasing [31].

Dynamically adjusting the frequency and voltage of memories (i.e. *memory DVFS*) has been suggested for server architectures by H. David et al. [50]. They observe that many of the SPEC CPU 2006 workloads require significantly less than the peak bandwidth. By

reducing the frequency and voltage for these workloads dynamically, the memory power is reduced by 10.4% in average with minimal performance impact. Due to limited control of supply voltage and frequency in the CACTI models, memory DVFS has not been investigated in this work. In CACTI, the memory bank V_{dd} is controlled by the size and the type of transistor technology that is selected. The potential for memory DVFS as a system knob should be explored in future work with memory-aware system scenarios.

6.3 CPU related

Intel SpeedStep is well suited for fast scenario switching. Its operating point can be controlled by software and a maximum of 10µs are required to do a switch [34]. Of the considered applications, the prime number checker is the application with the most frequent scenario switching. Even with this application, the scenario prediction and switching overhead is negligible. According to the calculations in Chapter 3, most of the operating points are Pareto-optimal, which is an advantage for DVFS applications. The reason why one of the operating points is not Pareto-optimal could be the quite simplifying assumptions that were made prior to the power calculations. If only the dynamic power of the processor is considered, all the operating points are Pareto-optimal. In [35] it is studied how Intel and AMD specify processor power and how the specifications can be used to compare processor power. TDP is described as a "worst-case value", and that a typical workload will dissipate less power than the rated TDP. It is specifically stated to not use TDP as a measure for processor power for this reason. AMD provides a different characteristic for their processors, called ACP (Average CPU Power), to give a more realistic impression of their processors' power consumption. The ACP of a processor is the geometric mean of the power it dissipates when running a set of benchmarks. ACP should therefore be a better suited measure for processor power than TDP. No ACP with frequency and voltage scaling has been found during this work, but this can be further investigated in future work.

Using TDP as CPU power is clearly an overestimation, but it seems to balance the high power consumption of the memories. The focus in this work is not to obtain very accurate power results. It is the dynamism in the applications and the methodology for exploiting it that is central, so as long as the numbers are not completely out of scale, the demonstrations should give good insight in the presented methodology. The intention of the methodology is to better explore how the possible power reductions in different parts of a system affect each other, and it is therefore the relative size of the numbers that is important for demonstrating the methodology. When using the developed methodology and algorithm in a real-life system scenario design process, the theoretical energy calculations should as much as possible be verified or replaced with measured values. There may be power variations even due to the application itself. By doing physical measurements of the actual platform, many sources of error are eliminated.

6.4 Methodology

The methodology is developed while trying to design system scenarios for the presented applications. The methodology should still be useful for a wide range of other applications,

due to the big differences between the considered applications.

The methodology should be applicable also for other kinds of dynamism, except the procedures that are CPU/memory specific. It can also be extended in order to deal with more than two-dimensional dynamism. For the steps in which each dynamism is considered individually, the computational complexity will only increase linearly with the added number of dimensions. In the exploration and exploitation step however, the complexity will increase exponentially with the added dimensions, since each added system knob adds an extra level to the nested for-loops in Algorithm 1 and 2. This is not critical unless the added system knobs have many possible settings. The methodology's complexity is linear in terms of adding more RTSs.

Exactly how a system will be used heavily influences the design of system scenarios, so some assumptions had to be made; e.g., concerning the time budget, when and how new inputs are received and possible corner case behavior. For simplicity no deadline is assumed initially, so that the focus is only on saving as much energy as possible. It is also assumed that the tasks are processed continuously, i.e. no waiting energy consumption between inputs. The results of this showed that energy reductions from slowing down the CPU can be outweighed somewhat or completely by the increased standby time of the memory. For HMMER the optimal balance between CPU and memory power was found to be at 1.2GHZ and 1.0GHz operation, where memory is responsible for 41-48% of the total energy given the optimal memory configuration. Later, to make the results more interesting and realistic, a deadline is introduced so that the most demanding task can just be finished in the highest performance scenario. When continuous input processing is still assumed (i.e. no waiting between inputs), the only effect is that some of the low-frequency operating points cannot always be used. When also assuming an expected period between each input, in which the CPU and memory have to remain idle while waiting for the next input, the lowest possible operating frequency is always optimal. This is because it reduces the total number of CPU cycles for each expected period.

When using DVFS in system scenario design, the results are often compared with *race to idle* (or race to halt) implementations. With race to idle, the CPU executes a task at maximum performance and enters a low-energy mode when finished, while waiting for the next task. By executing at full speed, the time spent in the low-energy mode is maximized. To motivate system scenario design, some papers however use a race to idle implementation where the CPU busy-waits between the tasks, which results in a high energy consumption to compare the system scenario design against. For applications where a responsive CPU is required this may be necessary, but it can also be unrealistic since many of today's CPUs provide very energy efficient low-energy modes. For the applications presented in this work, the CPU could have been put in a sleep mode when the processing of an input finishes before deadline. This may however not be feasible, as discussed, and no numbers could be found for Intel Pentium M low-power modes. As a compromise between this and leaving the CPU in a high performance state, the CPU's operating frequency is therefore reduced to 0.6GHz in the idle time. Changing the operating point when the processing is finished is not a product of the system scenario methodology, so this is also done in the static implementations that are used for comparison.

Chapter 7

Conclusion

In this thesis, system scenario development with multi-dimensional dynamism is explored and evaluated. More specifically, the DVFS optimizations of traditional system scenario methodology are combined with the recently developed memory-aware system scenario extension. Several papers present promising results of the system scenario methodology using CPU optimizations such as DVFS. For applications that require much memory however, the energy reductions from system scenario-based DVFS will be minimal if the total energy is anyway dominated by memory energy. Memory-aware system scenarios have been introduced with dynamically reconfigurable memories as a response to this problem. By utilizing both DVFS and dynamic memories, the dynamism in applications that are both memory and computationally intensive can be exploited. In this thesis, such a DVFS and memory-aware system scenario development is tested on four very different applications.

The first is a CPU intensive prime number checker with little memory requirements but very dynamic and hard to predict execution time. By delaying the scenario prediction however, the predictability of the execution time was increased and the scenario implementation is estimated to use 20.2% less CPU energy than the original. In these calculations, no power consumption is included from the potential waiting or idle time.

The next three applications are selected benchmarks from the SPEC CPU 2006 benchmark suite. These are characterized by being very intensive and dynamic in terms of both CPU and memory usage. The challenge is to exploit both of these dynamisms to save as much energy as possible without compromising performance. Through system scenario design, a number of RTSs and scenarios are identified and exploited, and up to 31% energy is saved. Datasheet numbers of an Intel Pentium M processor and CACTI memory models are used to estimate the energy savings. Memory dynamism is exploited both in terms of allocation timing and size. The system scenarios are designed both with and without deadlines. The complexity of the design process made it clear that a specialized methodology for identifying and exploiting dynamism in memory and CPU usage would be useful.

Such a methodology, which is slightly different from the traditional methodology, is therefore developed and tested on the four applications throughout this work. The presented methodology describes quite accurately how to detect CPU cycle and memory dynamism in applications through profiling and code inspection, and how to obtain the optimal combination of the different platform settings. Multi-dimensional dynamism is

a big challenge in system scenario design. The proposed methodology identifies and analyzes each dimension of the dynamism separately, and then combines it all in order to fully exploit the dynamism. This way, the complexity of dealing with a number of different variables is reduced during identification, and during exploitation one is able to rule out the sub-optimal solutions. The RTSs are not grouped into scenarios based on similar costs, but on optimal execution with the same operating point and configurations. The methodology is targeted at simple single-core platforms.

The feasibility of system scenario design for a given application does not only depend on the degree of dynamism among the identified RTSs, but on the expected occurrence of each RTS. This is because the most demanding RTSs typically will not benefit as much from the system scenarios as the less demanding RTSs. The algorithms presented in this work takes both the occurrence and requirements of the RTSs into account when optimizing the system scenarios. Demanding and/or frequent RTSs thus have more influence on the final set of system scenarios.

There is a considerable complexity involved in exploring and exploiting how the possible power reductions in different parts of a system affect each other. It is believed that the suggested methodology can be of help in this process. It is not meant as a replacement of the original methodology, but as an additional guidance in relevant situations.

Bibliography

[1] SPEC Open Systems Group. Spec cpu2006 run and reporting rules. `https://www.spec.org/cpu2006/Docs/runrules.html`. Accessed: 2015-09-23.

[2] Jed Scaramella. Worldwide server power and cooling expense 2006-2010 forecast. *IDC*, 1, September 2006.

[3] Thomas L. Martin. *Balancing Batteries, Power, and Performance: System Issues in CPU Speed-Setting for Mobile Computing.* PhD thesis, Carnegie Mellon University, August 1999.

[4] Shekhar Borkar and Andrew A. Chien. The future of microprocessors. *Commun. ACM*, 54(5):67–77, May 2011.

[5] Stefan Valentin Gheorghita. *Dealing with Dynamism in Embedded System Design: Application Scenarios.* PhD thesis, Technische Universiteit Eindhoven, 2007.

[6] Stefan Valentin Gheorghita, Martin Palkovic, Juan Hamers, Arnout Vandecappelle, Stelios Mamagkakis, Twan Basten, Lieven Eeckhout, Henk Corporaal, Francky Catthoor, Frederik Vandeputte, and Koen De Bosschere. System-scenario-based design of dynamic embedded systems. *ACM Trans. Des. Autom. Electron. Syst.*, 14(1):3:1–3:45, January 2009.

[7] P. Macken, M. Degrauwe, M. Van Paemel, and H. Oguey. A voltage reduction technique for digital systems. In *Solid-State Circuits Conference, 1990. Digest of Technical Papers. 37th ISSCC., 1990 IEEE International*, pages 238–239, Feb 1990.

[8] Antoni Portero, G. Talavera, M. Monton, Borja Martinez, F. Cathoor, and J. Carabina. Dynamic voltage scaling for power efficient mpeg4-sp implementation. In *Application-specific Systems, Architectures and Processors, 2006. ASAP '06. International Conference on*, pages 257–260, Sept 2006.

[9] Etienne Le Sueur and Gernot Heiser. Dynamic voltage and frequency scaling: The laws of diminishing returns. In *Proceedings of the 2010 International Conference on Power Aware Computing and Systems*, HotPower'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[10] David C. Snowdon, Stefan M. Petters, and Gernot Heiser. Accurate on-line prediction of processor and memory energy usage under voltage scaling. In *Proceedings of the 7th ACM &Amp; IEEE International Conference on Embedded Software*, EMSOFT '07, pages 84–93, New York, NY, USA, 2007. ACM.

[11] Vivek Tiwari, Sharad Malik, Andrew Wolfe, and Mike Tien-Chien Lee. Instruction level power analysis and optimization of software. *Journal of VLSI signal processing systems for signal, image and video technology*, 13(2-3):223–238, 1996.

[12] I. Filippopoulos, F. Catthoor, P.G. Kjeldsberg, E. Hammari, and J. Huisken. Memory-aware system scenario approach energy impact. In *NORCHIP, 2012*, pages 1–6, Nov 2012.

[13] Iason Filippopoulos, Francky Catthoor, and Per Gunnar Kjeldsberg. Exploration of energy efficient memory organisations for dynamic multimedia applications using system scenarios. *Design Automation for Embedded Systems*, 17(3-4):669–692, 2014.

[14] Karthik T. Sundararajan, Timothy M. Jones, and Nigel Topham. A reconfigurable cache architecture for energy efficiency. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF '11, pages 9:1–9:2, New York, NY, USA, 2011. ACM.

[15] Spec - standard performance evaluation corporation. `https://www.spec.org/`. Accessed: 2015-09-26.

[16] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006.

[17] Zhe Ma, Pol Marchal, Scarpazza, Daniele Paolo, Peng Yang, Chun Wong, José Ignacio Gómez, Stefaan Himpe, Chantal Ykman Couvreur, and Francky Catthoor. *Systematic Methodology for Real-Time Cost-Effective Mapping of Dynamic Concurrent Task-Based Systems on Heterogeneous Platforms*. Springer Netherlands, 2007.

[18] A. Agarwal and J. Lang. *Foundations of Analog and Digital Electronic Circuits*. The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science, 2005.

[19] Andrea Castagnetti, Cecile Belleudy, Sebastien Bilavarn, and Michel Auguin. Power consumption modeling for dvfs exploitation. In *Proceedings of the 2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, DSD '10, pages 579–586, Washington, DC, USA, 2010. IEEE Computer Society.

[20] M. Horowitz, T. Indermaur, and R. Gonzalez. Low-power digital design. In *Low Power Electronics, 1994. Digest of Technical Papers., IEEE Symposium*, pages 8–11, Oct 1994.

[21] K. Roy, S. Mukhopadhyay, and H. Mahmoodi-Meimand. Leakage current mechanisms and leakage reduction techniques in deep-submicrometer cmos circuits. *Proceedings of the IEEE*, 91(2):305–327, Feb 2003.

[22] Christopher J. Hughes, Jayanth Srinivasan, and Sarita V. Adve. Saving energy with architectural and frequency adaptations for multimedia applications. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 34, pages 250–261, Washington, DC, USA, 2001. IEEE Computer Society.

[23] Kihwan Choi, K. Dantu, Wei-Chung Cheng, and M. Pedram. Frame-based dynamic voltage and frequency scaling for a mpeg decoder. In *Computer Aided Design, 2002. ICCAD 2002. IEEE/ACM International Conference on*, pages 732–737, Nov 2002.

[24] Dongkun Shin and Jihong Kim. Optimizing intra-task voltage scheduling using data flow analysis. In *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, ASP-DAC '05, pages 703–708, New York, NY, USA, 2005. ACM.

[25] Akihiko Miyoshi, Charles Lefurgy, Eric Van Hensbergen, Ram Rajamony, and Raj Rajkumar. Critical power slope: Understanding the runtime effects of frequency scaling. In *Proceedings of the 16th International Conference on Supercomputing*, ICS '02, pages 35–44, New York, NY, USA, 2002. ACM.

[26] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, 5th edition edition, 2012.

[27] Christoph Lameter. Numa (non-uniform memory access): An overview. *Processors*, 11(7), August 2013.

[28] Philip Garcia, Katherine Compton, Michael Schulte, Emily Blem, and Wenyin Fu. An overview of reconfigurable hardware in embedded systems. *EURASIP J. Embedded Syst*, 2006.

[29] Angeliki Kritikakou, Francky Catthoor, Vasilios Kelefouras, and Costas Goutis. A scalable and near-optimal representation of access schemes for memory management. *ACM Trans. Archit. Code Optim.*, 11(1):13:1–13:25, February 2014.

[30] Aashish Phansalkar, Ajay Joshi, and Lizy K. John. Subsetting the spec cpu2006 benchmark suite. *Computer Architecture News*, 32(1), March 2007.

[31] T. K. Prakash and L. Peng. Performance characterization of spec cpu2006 benchmarks on intel core 2 duo processor,. *ISAST Transactions on Computers and Software Engineering*, 2(1):36–41, 2008.

[32] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, December 2007.

[33] J. Hamers and L. Eeckhout. Scenario-based resource prediction for qos-aware media processing. *Computer*, 43(10):56–63, Oct 2010.

[34] Intel. Enhanced intel speedstep technology for the intel pentium m processor. White paper, Intel Corporation, March 2004.

[35] Scott Huck. Measuring processor power - tdp vs. acp. White Paper Revision 1.1, Intel Data Center Group, April 2011.

[36] AMD. Acp - the truth about power consumption starts here. White paper, Advanced Micro Devices, Sunnyvale, California, 2009.

[37] L.P. Hewlett-Packard Development Company. Cacti. `http://www.hpl.hp.com/research/cacti/`, 2008. Accessed: 2015-10-27.

[38] David Tarjan (creator). Jung Ho Ahn (last update). Cacti 5.3 (rev 174). http://quid.hpl.hp.com:9081/cacti/detailed.y, June 2009. Accessed: 2015-10-27.

[39] Yahya H. Yassin, Per Gunnar Kjeldsberg, and Francky Catthoor. System scenario framework evaluation on efm32 using the h264/avc encoder control structure. In *The 22nd European conference on circuit theory and design, ECCTD2015*, Trondheim, Norway, August 2015.

[40] RotateRight. Software, tools and services. http://www.rotateright.com/, 2015. Accessed: 2015-10-09.

[41] RotateRight. Zoom quick start guide. http://www.rotateright.com/zoom-quick-start/zoom-quick-start.html, July 2013. Accessed: 2015-10-28.

[42] Valgrind Developers. Massif: a heap profiler. http://valgrind.org/docs/manual/ms-manual.html, 2015. Accessed: 2015-10-02.

[43] Valgrind Developers. Valgrind user manual. http://valgrind.org/docs/manual/cg-manual.html, 2015. Accessed: 2015-11-07.

[44] Inc Free Software Foundation. Gnu gprof. https://sourceware.org/binutils/docs/gprof/, November 2008. Accessed: 2015-10-05.

[45] S. R. Eddy. *HMMER User's Guide*. Howard Hughes Medical Institute and Dept. of Genetics, Saint Louis, USA, version 2.2 edition, August 2001.

[46] S. R. Eddy. Profile hidden markov models. *Bioinformatics*, 14(9):755–763, 1998.

[47] Accelrys Inc. Gcg manual - hmmercalibrate, hmmersearch. http://www.biology.wustl.edu/gcg/gcgmanual.html, January 2002. Accessed: 2015-10-13.

[48] Paul Berube. Additional fdo inputs - 181.mcf. http://webdocs.cs.ualberta.ca/~berube/compiler/fdo/inputs.shtml. Accessed: 2015-09-14.

[49] Srdjan Stanković, Irena Orović, and Ervin Sejdić. Digital video. In *Multimedia Signals and Systems*, pages 199–231. Springer US, 2012.

[50] *Memory Power Management via Dynamic Voltage/Frequency Scaling*, Karlsruhe, Germany, June 2011.

[51] T. M. Apostol. *Introduction to Analytic Number Theory*. Springer, 1976.

[52] Standard Performance Evaluation Corporation. Runspec avoidance. https://www.spec.org/cpu2006/Docs/runspec-avoidance.html, September 2011. Accessed: 2015-10-06.

[53] A.J. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *Information Theory, IEEE Transactions on*, 13(2):260–269, April 1967.

[54] Sriram Swaminathan, Russell Tessier, Dennis Goeckel, and Wayne Burleson. A dynamically reconfigurable adaptive viterbi decoder. In *Proceedings of the 2002 ACM/SIGDA Tenth International Symposium on Field-programmable Gate Arrays*, FPGA '02, pages 227–236, New York, NY, USA, 2002. ACM.

[55] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi. Cacti 5.1. Technical Report HPL-2008-20, HP Laboratories, Palo Alto, April 2008.

Appendix A

Miscellaneous

A.1 Dynamism in memory accesses

If the *use* of allocated memory is dynamic, and it can be predicted that a set of allocated memory locations will not be used in a long time, these memory locations can be put to sleep for a while. This kind of dynamism is harder to find than the dynamism in amounts of allocated memory. However, a common example is where a big chunk of memory is traversed sequentially by the application. Then it can be beneficial to partition this memory into banks and let the parts that are currently not accessed sleep. To identify these opportunities, look for very low cache miss rates and big memory footprint when profiling. Relatively long execution time is also necessary to make it feasible to exploit. Also, code inspection and good knowledge of how the application works should be useful.

The memory banks must be awakened with just enough time to get ready before being accessed. From the currently accessed memory location, an address offset can be used to determine which bank must be activated (Figure A.1). This offset will depend on the memory's wake-up time and the execution speed. The execution speed depends on the RTS and the CPU frequency. Therefore, for every change of RTS, the offset must be updated to match the new RTS. If the scenario implementation includes frequency scaling, the offset must also be updated after a frequency change.

Efficient caching mechanisms already exploit such memory access patterns, and in a much more flexible way. Memory architectures that use scratchpads (see Section 2.4) instead of caches are more likely to benefit from the suggested partitioning.

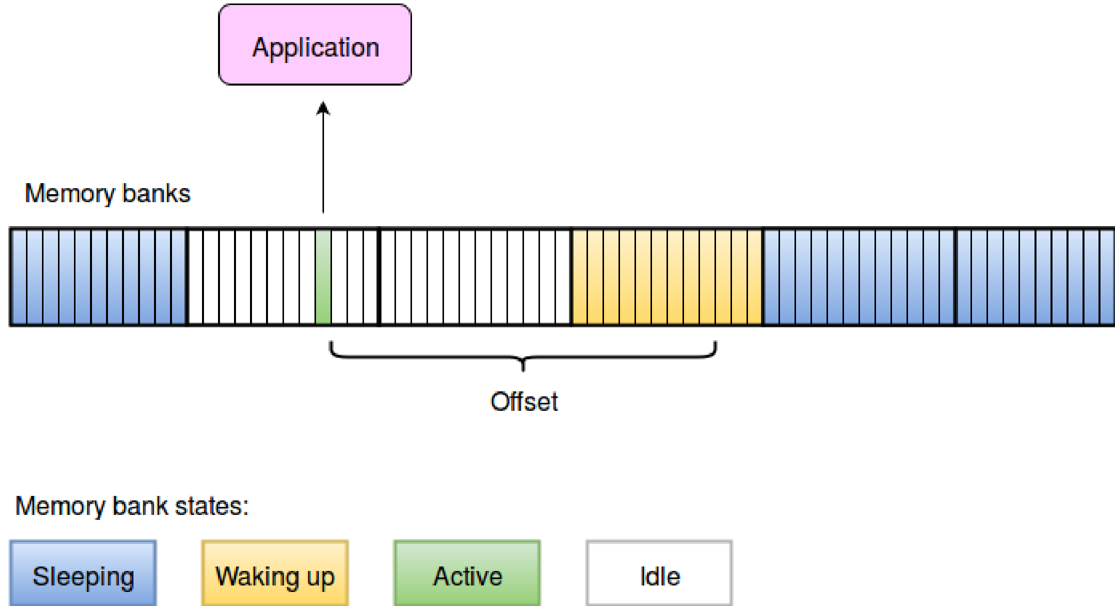


Figure A.1: Main memory partitioned into memory banks. A memory address offset added to the currently accessed address is used to wake up memory banks before they are used.

A.2 Prime number probabilities

Let $\pi(x)$ denote the number of primes up to a number x where $x > 0$. Then,

$$\lim_{x\to\infty} \frac{\pi(x)\ln(x)}{x} = 1 \tag{A.1}$$

where ln() is the natural logarithm. This is known as the Prime Number Theorem, which was independently proved by Jacques Hadamard and Charles Jean de la Vallee-Poussin in 1896[51]. By rewriting Equation A.1, the function $\pi(x)$ for the number of primes less than a number x is obtained:

$$\pi(x) \sim \frac{x}{\ln(x)} \tag{A.2}$$

The probability of choosing a prime out of all numbers in the range 0 to x is therefore:

$$P(prime) \sim \frac{1}{\ln(x)} \tag{A.3}$$

The probability of *not* choosing a prime in the range 0 to 637,538,053 (the maximum number input to the prime number checker in Section 5.1) is then:

$$P(notPrime) \sim 1 - \frac{1}{\ln(637538053)} = 0.951 \tag{A.4}$$

In the system scenario implementation with the original prime checker code, this is the probability for mispredicting a scenario. The code changes performed during the scenario

development does however reduce this drastically. Just by checking if the input number is dividable by two or three before predicting scenario, the probability for scenario misprediction is reduced to 0.28:

$$0.951 - 0.50 - 0.50 \cdot 0.33 = 0.28 \quad \text{(A.5)}$$

A.3 Controlling the benchmarks

This section describes what is done to be able to control and modify the benchmarks.

- Turn off the checksum (`check_md5=0`) in the SPEC build config-file.
- Issue `--fake` run commands as described in [52].
- Extract run commands from the generated logs.
- Make with options:
  `-I. -std=c99 -DSPEC_CPU -DNDEBUG -O2 -fno-strict-aliasing -DSPEC_CPU_LP64 -fno-omit-frame-pointer -g`

  `-fno-strict-aliasing` in order to get complete callstacks. Thread time profiling is not affected and performance of most code is not considerably impacted [41].
- Run the generated executable according to the extracted run commands.

A.3.1 How to run HMMER

The following command executes `hmmersearch` according to the benchmark suite:
`./hmmer_executable nph3.hmm swiss41`

`hmmercalibrate` is executed with the following command:
`./hmmer_executable --fixed 0 --mean 500 --num 500000 --sd 350 --seed 0 retro.hmm`

Only `hmmersearch` is investigated in this work, but there should be a similar dynamism present in `hmmercalibrate` since it uses `P7Viterbi()` in a similar way. The options explained below are not necessary, but can be used to replace the default values. More options and information can be found in the HMMER User's Guide [45].

`--fixed` fixes the length of the random sequences used in the calibration to n. The default is to generate sequences with variable lengths from a Gaussian distribution.

`--set` sets the random number seed to a positive integer to get equal results of two `hmmcalibrate` runs with same input HMM.

`--num` sets the number of synthetic sequences. Higher value gives better accuracy. The default is 5000, which is experimentally selected trade-off between computation time and accuracy.

A.3.2 How to run MCF

The following command executes the application according to the benchmark suite:
`./mcf_executable inp.in`

A.3.3 How to run H.264

The following command executes the application according to the benchmark suite:
`./h264_executable -d foreman_test_encoder_baseline.cfg`

A.4 MCF data set generator

The MCF data set generator [48] made by Paul Berube is used to generate additional MCF inputs. The number of timetabled and dead-head trips can be specified to the generator. Giving the same specifications to the generator results in the same MCF input being produced. The generator is intended for the SPEC CPU2000 version of the MCF benchmark, but according to [16] there are minimal changes to the MCF benchmark.

A.5 Viterbi algorithm

The Viterbi algorithm [53] is a dynamic programming algorithm for finding the most likely sequence of hidden states (called the Viterbi path) which results in a sequence of observed events. It was proposed by Andrew Viterbi in 1967 as an algorithm for decoding convolutional codes over noisy digital communication links. The algorithm is commonly used for encoding and decoding convolutional codes [12], with appliances like CDMA, GSM, dial-up modems, satellite, deep-space communications and 802.11 wireless LANs. A more recent application is within speech recognition, where the observed sequence of events is the acoustic signal, and a string of text is considered as the hidden cause of the acoustic signal.

The Viterbi algorithm is also widely used to decode convolutional codes applied to overcome data corruption in digital communication channels. Swaminathan et al. [54] implements a dynamically reconfigurable Viterbi decoder. Changing channel noise conditions are exploited by run-time dynamic reconfiguration of hardware, leading to a performance improvement of 20% (in terms of processed bandwidth). The amount of computation performed and the amount of storage used at both fine- and coarse-timescale level. Reconfiguration based on each received symbol is infeasible with current FPGA technology.

Filippopoulos et al. [12] uses a Viterbi encoder to demonstrate the usefulness of memory-aware scenario development. By adapting the number of active memory banks according to the SNR of a signal (which determines the necessary state sequence length), Filippopoulos et al. shows a energy reduction of more than 70% in situations with low noise.

A.6 CACTI

For the memory banks up to 32MB, CACTIs normal cache interface is used. For the bigger memory banks the CACTI Pure RAM interface is used. Table A.6 and A.6 shows the required input parameters for these models, and the selected values.

In the CACTI 5.1 Technical Report [55] the different transistor types are explained. The technology modelling in CACTI 5 is based on the ITRS (International Technology

Table A.1: CACTI cache parameters.

Cache Size (bytes)	Up to 33554432
Line Size (bytes)	64
Associativity	1
Nr. of Banks	1
Technology Node (nm)	40

Table A.2: CACTI RAM parameters.

RAM Size (bytes)	Between 536870912 and 33554432
Nr. of Banks	1
Read/Write Ports	1
Read Ports	1
Write Ports	0
Single Ended Read Ports	0
Nr. of Bits Read Out	512
Technology Node (nm)	40
Temperature (300-400 K, steps of 10)	300
All Transistor Types	ITRS-LOP
Interconnect projection type	Aggressive
Type of wire outside mat	Global

Roadmap for Semiconductors) roadmap. ITRS define different transistor types such as HP (High Power), LSTP (Low Standby Power) and LOP (Low Operating Power) that can be used in CACTI. The default is HP, which are state-of-the-art fast transistors that tend to be quite leaky. In this work, the LOP type have been used instead for the memory banks bigger than 32MB in order to reduce standby power. LOP use considerably lower supply voltage than HP.