Collecting signatures to model latency tolerance in high-level simulations of microthreaded cores

Irfan Uddin, M.; Jesshope, C.R.; van Tol, M.W.; Poss, R.

DOI
10.1145/2162131.2162132

Publication date
2012

Published in
RAPIDO '12: proceedings of the 2012 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools

Collecting signatures to model latency tolerance in high-level simulations of microthreaded core

M Irfan Uddin, Chris R. Jesshope, Michiel W. van Tol, Raphael K. Poss
Computer Systems Architecture group, Institute for Informatics University of Amsterdam
Sciencepark 904, 1098 XH Amsterdam The Netherlands
{mirfanud, c.r.jesshope, mwvantol, r.c.poss}@uva.nl

ABSTRACT

The current many-core architectures are generally evaluated by a detailed emulation with a cycle-accurate simulation of the execution time. However this detailed simulation of the architecture makes the evaluation of large programs very slow. Since the focus in many-core architecture is shifting from the performance of the individual core to the overall behavior of chip, high-level simulations are becoming necessary, which evaluate the same architecture at less detailed level and allow the designer to make quick and reasonably accurate design decisions. We have developed a high-level simulator for the design space exploration of the Microgrid, which is a many-core architecture comprised of many fine-grained multi-threaded cores. This simulator allows us to investigate mapping and scheduling strategies of families (i.e. groups of threads) in developing an operating environment for the Microgrid. The previous method to evaluate the workload counted in basic blocks was inaccurate. The key problem is that with many concurrent threads the latency of certain instructions are hidden because of the multi-threaded nature of the core. This paper presents a technique to manage the execution time of different types of instructions with thread concurrency. We believe to achieve high accuracy in evaluating programs in the high-level simulator.

Categories and Subject Descriptors

D.2.8 [**Performance of Systems**]: Performance estimation—*Performance measures*

General Terms

Experimentation, Estimation, Performance

Keywords

Performance estimation, Automatic annotation

1. INTRODUCTION

The design of computer architecture is a time consuming process and therefore simulators are developed in the overall design process of the computer and its operating environment. These simulators are used for the evaluation of the computer architecture and appear in different formats and at different levels of detail. Different evaluation tools have different complexity, execution time and accuracy.

Typically processors are emulated using a detailed simulation which models all the features, such as instruction issue mechanisms, caches at all levels, load and store queues and branches. Although these simulators are very accurate, they are orders of magnitude slower than the real processors. The evaluation of architectures with these simulators requires the execution of a set of benchmarks, which can consist of billions of dynamically executed instructions, can take a long time, not to consider multiple executions of the benchmark. Moreover, simulations of architectures at the early stage of development with such a level of detail is not always desirable because of the considerable time and efforts to develop them. High-level simulation enables quick and reasonably accurate design decisions in the early stages of computer design. It complements detailed but slower architectural simulations, reducing total design time and cost.

This problem is further exacerbated in many-core architectures as software components can be space-shared in many different ways. In fact the greater the number of cores, the greater the combinatorial explosion in the number of configurations that need to be simulated in order to find the best solution [6].

To overcome the many drawbacks of detailed simulation of a processor, where detailed simulation is not only unnecessary but very much time consuming, high-level simulation techniques have been introduced to evaluate a processor at different abstraction levels. High-level simulations enable designers an early design space exploration with relatively little effort and time to develop a new tool. These types of simulations are commonly used in embedded systems to model performance and power. Some of the high-level simulation techniques are used by, Meyer et al. [12], Paul et al. [17] and Pimentel et al. [18]. The current trend of many-core systems makes high-level simulations more desirable, because the focus is shifting from the performance of the individual core to the overall behavior of the chip, where creation, communication and synchronization of concurrency

plays an important role. Some of the high-level simulation techniques in the general purpose processors are used by Eeckhout et al. [3] and Smith et al. [15].

It is very difficult to maintain accuracy in high-level simulations in conventional processor architectures where instruction execution time may vary depending on the instruction type and the state of the processor. For example in a many-core processor, a memory instruction may take as little as a single cycle if the data is located in L1 cache, or thousands of cycles if the request has to go off chip and there is contention from the many cores for the memory channel. This problem is further exacerbated in a multi-threaded processor design, where multiple threads may be able to hide the latency of some of these long latency instructions, so that a 1000-cycle latency on a memory request may appear to execute in a single cycle.

This paper presents a technique for classifying and identifying different instruction types based on their latency and aggregating them into a signature that can then be used together with the dynamic state of the processor to more accurately estimate the processor's load. This technique is applied to the SVP many core Microgrid [8], which uses a multithreaded processor with data-flow synchronization and is able to tolerate latencies of up to 1000 cycles in a typical configuration. As long as we can track the number of active threads in the high-level simulation, the signature of a basic block can be used to improve the estimated execution time of that block. It should be noted that although this technique is applied to the multi-threaded processor in this paper, the technique can be also applied to modeling other processors with different mechanisms that tolerate the latency of instructions.

The rest of the paper is organized as follow. Section 2 gives some background to the microthreaded architecture and the high-level simulator of the same architecture. It also presents the problems in the previous approach to high-level simulation, which motivates us to present a new technique in Section 3. We present some results in Section 4 to show the importance of collecting signatures in a many-core, multi-threaded processor. We give some related work in Section 5 and conclude in Section 6

2. MOTIVATION AND BACKGROUND

The Microgrid is a general many-core architecture developed at the University of Amsterdam which implements hardware multithreading using data flow scheduling and a concurrency management protocol in hardware to create and synchronize threads within and across cores on chip [8]. The suggested concurrent programming model for this chip is based on fork-join constructs where each created thread can define further concurrency hierarchically. This model is called SVP and is also applicable to current multicore architectures [21].

In our work, we focus on a specific implementation of the Microgrid architecture where each core contains a single-issue, in-order RISC pipeline with an ISA similar to DEC/Alpha, and all cores are connected to a on-chip shared memory network [9, 2]. Each core implements the SVP actions in its instruction set and is able to support hundreds of threads and their contexts, called microthreads and tens of families (i.e. groups of microthreads) simultaneously. In the rest of the paper a thread means microthread, and a core means microthreaded core unless stated otherwise. To program this implementation, we use a system-level language called SL which integrates the SVP concurrency constructs as language primitives. In the rest of this paper, we use the shorthand *SVP program* to designate programs written in any language that supports the SVP protocol.

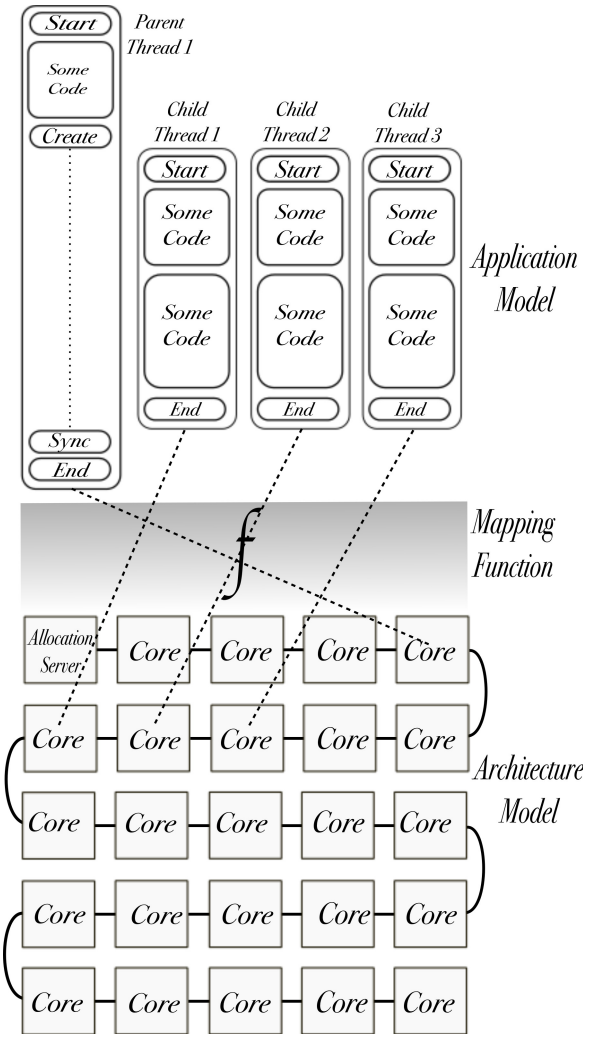

Figure 1: High-level simulation of the microthreaded architecture

The high-level simulator of SVP many-core systems [13] is developed to make quick and reasonably accurate design decisions in the evaluation of microthreaded architecture using multiple runs of benchmarks which can consist of billion of instructions executions. It also allows us to investigate mapping strategies of families to different cores in developing an operating environment for the Microgrid.

The high-level simulator provides us a simple way to model the microthreaded cores in large-scale system, avoiding all the details of executing instruction streams of a thread and focus more on mapping, scheduling and communication of

threads and families. It is not a replacement of the cycle-accurate simulator of the Microgrid, which emulates the microthreaded architecture with a detailed simulation of the execution time of individual components, rather it is a tool which is added for the evaluation of benchmarks for the same architecture but at a different level of abstraction, which is significantly faster at simulating the execution of SVP programs.

Figure 1 shows the high-level simulator of microthreaded architecture. The architecture model simulates the Microgrid where cores are used only to execute the workload of an instruction stream with no detailed simulation of pipeline, instruction issue mechanism or load and store queues. One of the cores is designated as the allocation server, which allocates core(s) when requested by the application model. In the middle we have mapping function which decides the allocation of families to cores and their scheduling. The application model is represented in the form of an SVP program. Compared to the cycle-accurate simulator where threads can interleave at every cycle showing a fine-grained interleaving of threads, the high-level simulator evaluates threads based on some time step showing a discrete event simulation of the workload of threads. Time step is computed to have the longest possible step in the execution time between synchronizing events over all executing threads and is explained in [13].

In the high-level simulator [13] we reduced the execution time of benchmarks to some hours compared to days required using the cycle-accurate simulator for the same architecture. The main focus of the initial implementation of the high-level simulator was performance improvements. The next step is to improve the accuracy of the high-level simulator compared to the cycle-accurate simulation. To quote Eeckhout *"highly accuracy performance estimates are illusory anyway, given the knowable level of design details"* [3], high-level simulations in general are less accurate. In this paper we are presenting a technique which will improve the accuracy of the high-level simulations to allow the designers to evaluate an architecture.

The high-level simulation of the Microgrid is based on the idea of discrete event simulations [14] and it uses a performance estimation of basic blocks to take discrete time steps in the simulation time. One of the performance estimation techniques is annotating the control flow graph (CFG) [1] in the source program with information useful to derive a cycle-accurate performance model. We used this annotation of CFG in the high-level simulation of the Microgrid but we divide the basic block into two parts: a) The part of the basic block which consists of instructions that do computation and are called non-SVP instructions. b) The part of the basic block, which does not do any computation but takes care of concurrency management and are called SVP instructions (e.g. create, sync, read and write from shared channel). The concurrency management statements are sent as an event along with its workload to perform these operations in the high-level simulator, we are more interested in estimating the performance of that part of the basic block which consist of only computation. In the rest of the paper we uses the term "basic block" to refer to the non-SVP instructions.

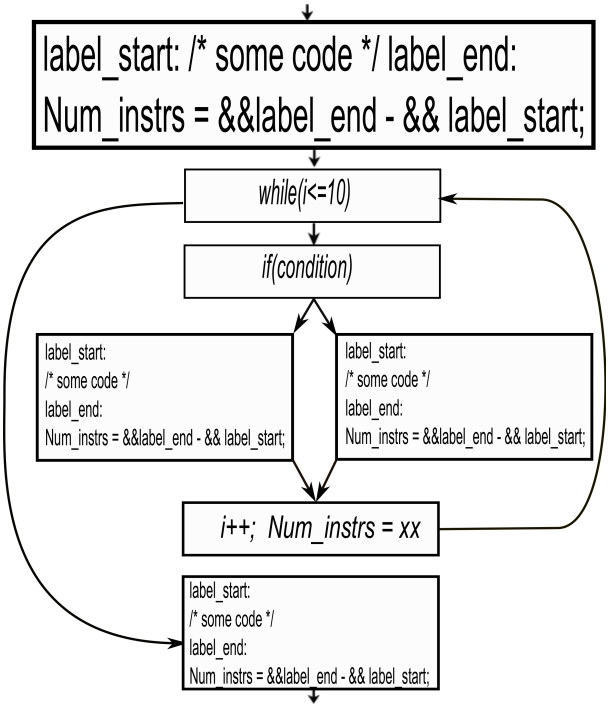

Figure 2: Performance estimation of a basic block by measuring the distance between the beginning and ending of the basic block

In the original implementation of high-level simulator, we place a label at the beginning and end of the basic block to compute the number of instructions between them when the program is compiled. Because we are using a RISC instruction set, we assume that every instruction will take one cycle. This means that the number of instructions we have in a basic block is actually its workload to perform the computation. But the high-level simulator runs on the host machine (i.e. x86) and we compute a calibration factor of the SVP program by executing it on the cycle-accurate simulator whose ISA (i.e. DEC/Alpha) is different and multiply that factor to the workload in the basic block. Figure 2 shows a CFG of an SVP program between two SVP instructions. It shows how labels are placed at the beginning and end of the basic block and how the number of instructions are counted from these labels. Although there might be multiple threads which are executing the same code, and their execution is interleaved, this instruction counting provides the workload of a basic block per thread. This technique has a few drawbacks:

- It counts instructions of basic blocks on the host machine (x86) and assign them workload to be processed on the high-level simulator which is simulating microthreaded architecture (DEC/Alpha).
- For complex and dynamic code we can not predict a calibration factor until execute the same trace on the cycle-accurate simulator.
- The calibration factor is used to statically estimate the average number of cycles to execute an instruction, which is inaccurate because the number of cycles

depends on the type of instruction and number of active threads. A long latency instruction (e.g. floating point) may take few cycles to complete with few concurrent threads, but with many concurrent threads the execution time can be reduced to single cycle (*cf* Section 1).

- Input programs have to be annotated manually. It is difficult to find basic blocks in a program without building an abstract syntax tree, which we want to avoid. Because we want the high-level simulator to focus more on the mapping of threads rather than analyzing programs. This approach works for a small program but considering large benchmarks this is not practical.
- It is difficult to count instructions executed inside conditionals.

Because of the multi-threaded nature of the core [2] instruction throughput is one cycle per instruction for all operations provided the sufficient number of active threads. However with a single thread the throughput is limited by the instruction latency. This motivates us to use signatures which is actually the instruction mix of a basic block and allow us to adapt the instruction throughput according to the number of concurrent threads at any time during the execution in the high-level simulator. Signatures are derived from [7] in the design space exploration of embedded systems.

This technique overcomes the drawbacks of the previous approach as follows:

- Signatures consist of the instruction set of microthreaded architecture (i.e. DEC/Alpha) and then evaluated by the high-level simulator which simulates the same architecture.
- We do not need any calibration factor.
- Signatures allow us to hide latency of long latency instructions when more threads are active.
- Input programs are automatically annotated with signatures of basic blocks. We are not building the abstract syntax tree; rather, we are using some string manipulation techniques.
- Instructions executed inside conditionals are counted.

3. AUTOMATED PROGRAM ANALYSIS

In the high-level simulation framework we have developed a tool, which is called the basic block analyzer to automatically analyze SVP programs and to generate signatures for all the basic blocks. The process of collecting signatures and annotating programs with signatures is shown in Figure 3. In the figure the part of a program between two SVP instructions is shown which consist of few basic blocks whose execution time is determined at run time. The source program is forwarded to the basic block analyzer which performs multiple iterations, analyzes the generated assembly code and adds the signatures of basic block to the source program. Some steps involved in the basic block analysis process are explained below.

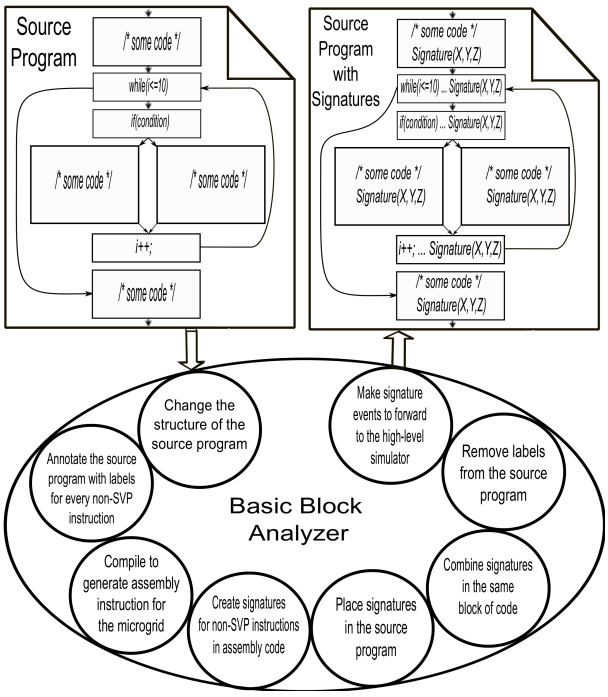

Figure 3: Basic block analyzer for SVP programs

```
sl_def(fibo_compute, void, sl_shparm(INT, prev),
    sl_shparm(INT, prev2), sl_glparm(INT*, fibo))
{
  sl_index(i);
__asm__ volatile("NORMAL_LABEL_8:");
  INT n = sl_getp(prev) + sl_getp(prev2);
__asm__ volatile("NORMAL_LABEL_9:");
  sl_setp(prev2, sl_getp(prev));
__asm__ volatile("NORMAL_LABEL_10:");
  sl_setp(prev, n);
__asm__ volatile("NORMAL_LABEL_11:");
  sl_getp(fibo)[i] = n;
__asm__ volatile("NORMAL_LABEL_12:");
}
sl_enddef
```

Listing 1: SVP program annotated with labels. (Only part of the program is shown for illustration)

```
...
NORMAL_LABEL_8:
        .set    nomacro
        addq $d1,$d0,$l1
        .set    macro
NORMAL_LABEL_9:
        mov $d0, $s1
NORMAL_LABEL_10:
        mov $l1, $s0
NORMAL_LABEL_11:
        .set    nomacro
        s8addq $l0,$g0,$l0
        stq $l1,0($l0)
        .set    macro
NORMAL_LABEL_12:
...
```

Listing 2: Assembly instructions of the microthreaded architecture for the Fibonacci program

3.1 Input program structure

Although it is possible to change the structure of the source program by building the abstract syntax tree and use the grammar of the language to transform it into the desired format, because of the development time and the complexity, we want the input programs to be in a format which is acceptable to the basic block analyzer i.e. every basic block in an SVP program is separated with braces and on separate line.

3.2 Input program annotation

The basic block analyzer places labels for every non-SVP instructions to identify when to create signatures in the assembly code. We consider an example program to calculate Fibonacci numbers in SVP. The program with labels for every non-SVP instruction is shown in Listing 1.

3.3 Compilation to assembly

The annotated input program is compiled to generate assembly code for the microthreaded architecture using SVP compiler. An example assembly code is shown in Listing 2 which contains the same labels from the input program. We start counting instruction and keep adding them to signature until we encounter the next label, at which point we add the signature to the source program and create a new signature.

```
sl_def(fibo_compute, void, sl_shparm(INT, prev),
    sl_shparm(INT, prev2), sl_glparm(INT*, fibo))
{
  sl_index(i);
signature[0] = 0; signature[1] = 0; signature[2] = 0;
__asm__ volatile("NORMAL_LABEL_8:");
  INT n = sl_getp(prev) + sl_getp(prev2);
signature[0] = 1; signature[1] = 0; signature[2] = 0;
__asm__ volatile("NORMAL_LABEL_9:");
  sl_setp(prev2, sl_getp(prev));
signature[0] = 1; signature[1] = 0; signature[2] = 0;
__asm__ volatile("NORMAL_LABEL_10:");
  sl_setp(prev, n);
signature[0] = 1; signature[1] = 0; signature[2] = 0;
__asm__ volatile("NORMAL_LABEL_11:");
  sl_getp(fibo)[i] = n;
signature[0] = 1; signature[1] = 0; signature[2] = 0;
__asm__ volatile("NORMAL_LABEL_12:");
}
sl_enddef
```

Listing 3: SVP program annotated with labels and signatures

3.4 Count assembly instructions

Every time the assembly code of a thread is analyzed an instance of the signature is created, which is a vector of three elements representing single latency instructions, fixed latency instructions and variable latency instructions at indexes 0, 1 and 2 respectively. The categorization of assembly instructions of the microthreaded architecture into abstract instruction set (AIS) is shown in Table 1.

AIS_SINGLE_LATENCY are those instructions, which take one cycle to complete. AIS_FIXED_LATENCY instructions take a fixed number of cycles to complete, however these cycles can be overlapped with other instructions when there are enough concurrent threads and therefore the cycles can be reduced to one cycle. AIS_VARIABLE_LATENCY can take many cycles depending on the access from different level of cache or off-chip memory. Also in this case the cycles to complete can be reduced to few or one cycle by their execution with other instructions.

Index	Abstract Instruction Set	Mnemonic	Cycles
0	AIS_ SINGLE_ LATENCY	Every instruction except in the two categories below and SVP instructions	1
1	AIS_ FIXED_ LATENCY	ADD[F,G,S,T]	3
		SUB[F,G,S,T]	3
		MUL[F,G,S,T]	3
		DIV[F,G,S,T]	8
		SQRT[F,G,S,T]	10
		MUL[L,V,Q]	3
		DIV[L,V,Q]	3
		UMULH	3
		BEQ, BGE, BGT, BLBC, BLBS, BLE, BLT, BNE, BR, BSR, JMP, JSR, RET MB, FETCH, EXCB, TRAPB, WMB	2
2	AIS_ VARIABLE_ LATENCY	LD[BU, WU, L*, Q*, S, T, G, F]	<~1000

Table 1: Categorization of instruction set of the Microgrid

The basic block analyzer counts the instruction mix between two labels, and places them into the signature vector. Before counting the instructions from the next label, it places signature back in the original program where the current label has appeared, and starts creating new signature from the next label. After all labels are visited and their signatures are collected, the input program has signatures for all the non-SVP instructions. An example code is shown in Listing 3.

3.5 Remove labels and combine signatures in the same basic block

We can remove the labels from the input program, because they are not used any more. We can have a basic block, which might consist of multiple signatures. These signatures can be combined, because they generate an event to the high-level simulator to send information of the basic block. Sending an event blocks the execution of the application model until registered in the architecture model, therefore it is more efficient if we combine multiple signatures in the same basic block into one signature which results into one event.

```
sl_def(fibo_compute, void, sl_shparm(INT, prev),
    sl_shparm(INT, prev2), sl_glparm(INT*, fibo))
{
  sl_index(i);
  INT n = sl_getp(prev) + sl_getp(prev2);
  sl_setp(prev2, sl_getp(prev));
  sl_setp(prev, n);
  sl_getp(fibo)[i] = n;
  Signature(4,0,0);
}
sl_enddef
...
```

Listing 4: SVP program with signature events

3.6 Loops

Loops are special cases, where basic blocks inside the loop execute multiple times and is determined at run time. We

Applications	Instructions executed in the cycle-accurate simulator	Instructions (non-SVP) executed in the high-level simulator	Error (%)
FFT	316220	305947	3,24
GOL	22461	20049	10,73
Mandelbrot	44456	42545	4,30
MergeSort	5498191	3988288	27,46
Smooth	703782	670404	4,7
Livermore (Kernel-23)	62928	511051	16.98

Table 2: Comparison of instructions executed in the high-level simulator to instructions executed in the cycle-accurate simulator

do not combine signatures of the whole loop, rather collect signatures for all basic blocks inside the loop. When the loop actually executes, these events are send to the high level simulator which keeps combining signatures until the loop terminates and then simulates their workload.

3.7 Make signature events

Signatures are collected to provide the workload of basic blocks to the high-level simulator which receives everything from the SVP program as an event. We convert the signature vector to signature event, which is basically a function call to the interface of the high-level simulator to provide some information. An SVP program with signature events is shown in Listing 4 and it is in the original format with the addition of information of basic blocks.

4. RESULTS

4.1 Accuracy of basic block analyzer

The basic block analyzer collects signatures for basic blocks in any SVP program. To validate the instruction counting by the basic block analyzer we compare the total signatures, which are actually the number of instructions counted in an SVP program with the number of instructions executed in the cycle-accurate simulator. Table 2 shows the results of some benchmarks executed on both simulators. Basic block analyzer only counts instructions for the non-SVP instructions, and cycle-accurate simulator shows the total instructions executed which includes both SVP and non-SVP instruction, therefore we have some error in the basic block counting. Moreover, basic block analyzer does not annotate library functions calls e.g. in case of merge sort the error is very high, because of calling some library functions. This motivates us to consider either annotating library functions with signatures or implement them as abstract components in the high-level simulator in the future work.

4.2 Importance of Signatures

4.2.1 Merge Sort

This experiment shows the multi-threaded nature of a core which allows us to hide latency of instructions. We are using only one core because we are interested in the accuracy of the load calculation for a single core considering the number of threads executing. Figure 4 shows the execution of Merge

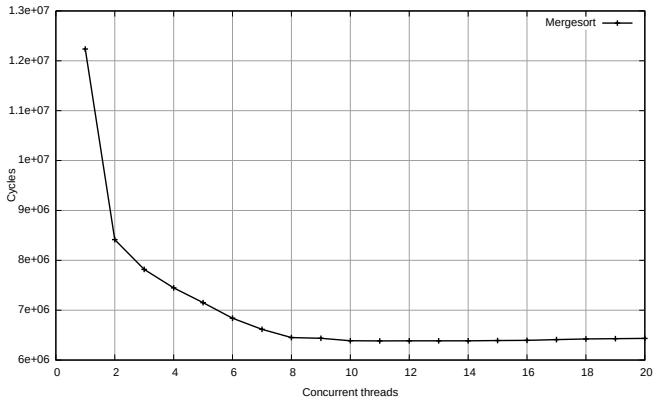

Figure 4: Merge sort executed on single core with different window sizes

sort in the cycle-accurate simulator using one core with different number of concurrent threads. With only one thread the performance is worst, but as the number of concurrent threads increases the microthreaded architecture hides latency for long latency instructions and therefore the performance increases. This paper presents the technique to collect signatures which can be used to adapt the throughput of the program based on the number of concurrent threads at any time of the execution of the program in the high-level simulator.

4.2.2 Uniform signatures

This experiment shows how different classes of instructions execute in the microthreaded architecture. Table 3 shows input parameters of this experiment. We create three different families of 100 threads each with instructions of only one class and execute them with window size 1 to 20. Window size is a value configurable at run-time for each family of threads, which defines the number of logical threads. These get mapped by the hardware onto hardware threads, whose count does not change. *Family_single* consists of only single latency instructions, *Family_fixed* consists of only fixed latency instructions and *Family_unbounded* consists of only variable latency instructions.

The execution of these three types of families in the cycle-accurate simulator is shown in Figure 5.

- The performance of *Family_single* remains the same irrespective of the number of concurrent threads, because single latency instructions takes one cycle to complete. With the window size of one, a new thread is created only when the entry for the already created thread is cleaned up and this overhead affect the performance.
- *Family_fixed* is performing badly when there is only one thread active, but as the number of threads increases its performance increases by hiding latency until it reaches the point where these instructions take only one cycle to complete.
- *Family_variable* is the worst in performance with exe-

Signature [AIS_SINGLE, AIS_FIXED, AIS_VARIABLE]	Number of threads	Window size
Family_single = [9,0,0]	100	1 - 20
Family_fixed = [0,9,0]	100	1 - 20
Family_variable = [0,0,9]	100	1 - 20

Table 3: Three families are created, where every family executes instructions of only one AIS

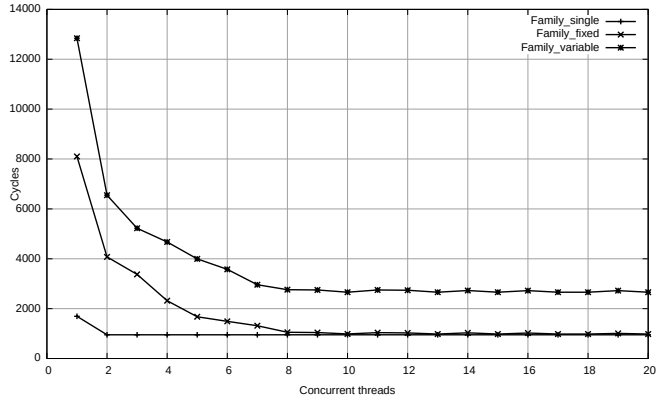

Figure 5: Simulated time of three families each consisting of only one type of instruction

cuting single thread because of accessing off-chip memory. But as the number of concurrent threads increases, the performance increases. The latency of variable latency instructions depends on different factors (e.g. access of memory from different levels of cache or off-chip memory) and therefore in some cases the latency can be tolerated to a single cycle. This experiment motivates us to investigate the behavior of different variable latency instructions on the microthreaded architecture and try to implement similar behavior in the high-level simulator in the future work.

5. RELATED WORK

The high-level performance estimation is an important factor in the fast embedded system design cycle. However, it is not trivial to get such an estimate without detailed implementation. In [1] performance estimation is used in both source-based and object-based to annotate the code with timing and other execution related information e.g. memory accesses and compare their execution with the cycle-based processor models. In [5], a source-based estimation technique is presented using the idea of *Virtual instructions* which are very similar to our abstract instruction set, but are directly generated by a compiler framework. Software performance is then calculated based on the accumulation of the performance estimates of these virtual instructions. In [3], a performance modeling approach is used for statistical simulation of the micro-architecture. Their simulation estimates the performance of programs with more details such as pipeline and cache behavior, while we address system-level modeling at higher level of abstraction.

The work presented in this paper is derived from the signatures used in the design space exploration of embedded systems by Pimentel et al. [7] to estimate the performance of processes using Kahn Process Network model of computation. Low level instructions are categorized into an abstract instruction set to estimate the performance of a process and enable the processes to execute at any architecture model. They generate instruction traces of the process and then using signatures to determine the performance while we use signatures at the time when the program is actually executing. We categorize low level instructions into an abstract instruction set to collect signatures which contains information of the microthreaded architecture and enable the high-level simulator to simulate the same instruction set on any machine. Moreover the signatures we collected does not give any information about performance until it is actually executed. Because performance depends on signature and number of concurrent threads.

In fact a lot of work is already done in the performance estimation area [20, 11, 4, 10, 16, 19]. The main difference between others and our work is that we do not have estimation of the basic block before it is actually executed because of the latency hiding. We collect signatures of basic blocks, and estimate the performance at the time of execution which depends on the type of instruction and the number of concurrent threads at that time, because of the multi-threaded nature of the core to hide latency in case of long latency instructions.

6. CONCLUSION

We have shown that the basic block analyzer counts the number of instructions in most cases with a high level of accuracy. Only in the case of code with library functions, which are not currently considered in this process reduces the accuracy. We also demonstrated the impact on performance of the number of concurrent threads using synthetic benchmarks. We showed that the three classes of instructions we identify have different execution characteristics based on the number of threads executing. This paper therefore introduced the techniques and motivation for dynamic signature-based modeling of processors at a high level where the processor is able to tolerate the latency of instructions. We are currently using this technique to build a more accurate high-level simulation of the Microgrid.

Acknowledgement

The author would like to acknowledge Andy Pimentel for feed back in this work.

7. REFERENCES

[1] J. R. Bammi, W. Kruijtzer, L. Lavagno, E. Harcourt, and M. T. Lazarescu. Software performance estimation strategies in a system-level design tool. In *Proceedings of the eighth international workshop on Hardware/software codesign*, CODES '00, pages 82–86, New York, NY, USA, 2000. ACM.

[2] K. Bousias, L. Guang, C. R. Jesshope, and M. Lankamp. Implementation and evaluation of a microthread architecture. *J. Syst. Archit.*, 55:149–161, March 2009.

[3] L. Eeckhout, S. Nussbaum, J. E. Smith, and K. D. Bosschere. Statistical simulation: Adding efficiency to the computer designer's toolbox. *IEEE Micro*, 23:26–38, September 2003.

[4] B. G., C. Brandolese, W. Fornaciari, F. Salice, D. Sciuto, and V. Trianni. An assembly-level execution-time model for pipelined architectures. In *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, ICCAD '01, pages 195–200, Piscataway, NJ, USA, 2001. IEEE Press.

[5] P. Giusto, G. Martin, and E. Harcourt. Reliable estimation of execution time of embedded software. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '01, pages 580–589, Piscataway, NJ, USA, 2001. IEEE Press.

[6] C. Grelck, K. Hammond, H. Hertlein, P. Holzenspies, C. Jesshope, R. Kirner, B. Scheuermann, A. Shafarenko, I. T. Boekhorst, and V. Wieser. Engineering Concurrent Software Guided by Statistical Performance. *Proc. Parco*, September 2011. (To appear).

[7] S. Jaddoe and A. D. Pimentel. Signature-based calibration of analytical system-level performance models. In *Proceedings of the 8th international workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation*, SAMOS '08, pages 268–278, Berlin, Heidelberg, 2008. Springer-Verlag.

[8] C. Jesshope. A model for the design and programming of multi-cores. *Advances in Parallel Computing*, High Performance Computing and Grids in Action(16):37–55, 2008.

[9] C. Jesshope, M. Lankamp, and L. Zhang. The implementation of an svp many-core processor and the evaluation of its memory architecture. *SIGARCH Comput. Archit. News*, 37:38–45, July 2009.

[10] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the 32nd annual ACM/IEEE Design Automation Conference*, DAC '95, pages 456–461, New York, NY, USA, 1995. ACM.

[11] S. Malik, M. Martonosi, and Y.-T. S. Li. Static timing analysis of embedded software. In *Proceedings of the 34th annual Design Automation Conference*, DAC '97, pages 147–152, New York, NY, USA, 1997. ACM.

[12] B. H. Meyer, J. J. Pieper, J. M. Paul, J. E. Nelson, S. Member, S. Member, S. M. Pieper, and A. G. Rowe. Power-performance simulation and design strategies for single-chip heterogeneous multiprocessors. *IEEE Trans. Comput*, 54, 2005.

[13] M.Irfan-Uddin, M. van Tol, and C. Jesshope. High level simulation of SVP many-core systems. *Parallel Processing Letters*, 21(4), December 2011. (To appear).

[14] J. Misra. Distributed discrete-event simulation. *ACM Comput. Surv.*, 18:39–65, March 1986.

[15] S. Nussbaum and J. E. Smith. Modeling superscalar processors via statistical simulation. In *in International Conference on Parallel Architectures and Compilation Techniques*, pages 15–24, 2001.

[16] C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Syst.*, 5:31–62, March 1993.

[17] J. M. Paul, D. E. Thomas, and A. S. Cassidy. High-level modeling and simulation of single-chip programmable heterogeneous multiprocessors. *ACM Trans. Des. Autom. Electron. Syst.*, 10:431–461, July 2005.

[18] A. D. Pimentel, C. Erbas, and S. Polstra. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE TRANSACTIONS ON COMPUTERS*, 55(2):99–112, 2006.

[19] A. Snavely, N. Wolter, and L. Carrington. Modeling application performance by convolving machine signatures with application profiles. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, pages 149–156, Washington, DC, USA, 2001. IEEE Computer Society.

[20] K. Suzuki and A. Sangiovanni-Vincentelli. Efficient software performance estimation methods for hardware/software codesign. In *Proceedings of the 33rd annual Design Automation Conference*, DAC '96, pages 605–610, New York, NY, USA, 1996. ACM.

[21] M. van Tol, C. Jesshope, M. Lankamp, and S. Polstra. An implementation of the SANE Virtual Processor using POSIX threads. *Journal of Systems Architecture*, 55(3):162–169, 2009. Challenges in self-adaptive computing (Selected papers from the Aether-Morpheus 2007 workshop).