Proceedings Work-in-Progress Session of LCTES 2012

June 12-13, 2012
Beijing, China

Edited by Jan Reineke

Message from the WiP Chair

Dear Colleagues:

Welcome to Beijing and to the Work-in-Progress (WiP) session of LCTES 2012. I am pleased to present to you six papers on WiP that describe innovative research contributions in the broad field of languages, compilers, tools and theory for embedded systems. The six accepted papers were selected from nine submissions.

The purpose of the LCTES WiP session is to provide researchers in academia and industry an opportunity to discuss their research ideas and to gather feedback from the community at large. Special thanks go to the Work-in-Progress Program Committee members Patricia Derler, Daniel Grund, Nan Guan, Claire Maiza, Hiren D. Patel, and Benny Åkesson for their good work in reviewing the submissions.

Jan Reineke
Work-in-Progress Chair
LCTES 2012

Program Committee

Patricia Derler	University of California, Berkeley, USA
Daniel Grund	Saarland University, Germany
Nan Guan	Uppsala University, Sweden
Claire Maiza	Verimag, France
Hiren D. Patel	University of Waterloo, Canada
Jan Reineke	Saarland University, Germany
Benny Åkesson	Eindhoven University of Technology, Netherlands

Table of Contents

Object Oriented Programming in C: A Case Study of TMS320C6418

Tanın Afacan

Aselsan A.Ş.
Ankara, Turkey
tafacan@aselsan.com.tr

Abstract

This paper presents an empirical study of the impact of Object Oriented Programming Implementation in C on the memory and execution time of a fixed-point Digital Signal Processor from Texas Instruments; TMS320C6418 [1]. Actually, the object-oriented approach introduces a significant performance penalty compared to classical procedural programming. One can find the studies of the object-oriented penalty on the system in terms of execution time and memory allocation in the literature. Since, to the author's best knowledge the study of the overheads of Object Oriented Programming implementation in C for the embedded systems is not widely published in the literature. Besides, it is possible to implement Object Oriented Design in a procedural language. The basic Object Oriented Programming features can be implemented in C such as creating objects, polymorphism, virtual functions, sub-classing, and inheritance. The main contribution of the paper is to bring further evidence that embedded system software developers have to consider the complexity and performance of Object Oriented Programming implementation in C in the embedded system programming. The results of the experiment show that Object Oriented Programming implementation in C adds significant complexity to the system, although it gives almost the same memory allocation and performance results as Object Oriented Programming implementation in an object-oriented language such as C++.

Keywords*: Digital Signal Processor (DSP), Object Oriented Programming (OOP), C, Object Oriented Design, C++*

1. Introduction

Object Oriented Programming (OOP) is a paradigm and methodology for software development and design [2] based upon the idea of breaking the complex software system down into its various objects, combining the data and the functions that operate on the designed entity. Even though the object-oriented approach is known to introduce a significant performance penalty compared to classical procedural programming [3], OOP has proved to be one of the major steps towards more productive and systematic software design [4]. In principle, any design could be realized without OOP methodology but in complex software projects the productivity and conceptual clarity of OOP typically far exceeds the traditional approaches [4]. Therefore, OOP has become very popular in the past years for software development and design.

As a consequence of the fast growing complexity and size of embedded systems, the requirements for embedded software development are changing [5]. The effort spent on developing the system becomes more important, compared to the per-unit cost of the device. It should become clear that OOP is a proper methodology for digital signal processor (DSP) programming [4]. In addition, OOP is a good methodology in DSP research and development as well.

Besides that, in embedded systems, ANSI C is the most commonly used language for DSP programming [6]. C has the advantages of high availability of compilers for wide range target processors, a well-deserved reputation for run-time efficiency [7]. Therefore, the embedded system software developers have not been very eager to adapt new technologies; especially the language adoption was very slow. Nevertheless, OOP is both a general methodology and a way of thinking, and a tool for programming [4]. It is possible to design and write programs based on OOP ideas without any specific OOP language, but if you choose correct language, you will be rewarded with a straightforward design and an even easier implementation [8]. Therefore, OOP features can be implemented in any programming language, but some languages are more suitable and flexible than others. One effective disadvantage of present OOP languages is the reduction in performance because of the object formalism. However, optimizing compilers keep trying to minimize or eliminate the overheads of the object-oriented languages [5].

Since C is commonly used in micro-controllers and it is possible to implement OOP in C, most of the features of OOP can be implemented via some techniques in C.

The goal of the paper is to supply the empirical memory allocation and performance data, while the performance is usually the major concern [9], to help the embedded system software developers to consider OOP implementation in a procedural language such as C. The second goal of the paper is to discuss the effect of OOP implementation in C to the reliability of the embedded system software.

2. Experiment

The experiment is done on TMS320C6418 of a custom design board. TMS320C6418 is one of the new generation highest-performance processors. TMS320C6418 has the operating frequency of 500 MHz. "Optimization Level" option of the compiler is set to "None" and "Opt. Speed vs. Size" option of the compiler is set to "Size Critical".

For the experiment, two designs are examined. First design is Debounce example; a toggle button for a microwave is modeled, which has the simple relations and simple implementation also short code length. Second design is Queue/Cached Queue example that has relatively complex relations, complex implementation and longer code length.

Figure 1 and Figure 2 show the sample UML class diagrams of Debounce and Queue/CachedQueue examples [7]. These designs

are implemented in C. Related code listings can be found also in [7]. OOP features such as abstraction, inheritance, and polymorphism can be implemented in a number ways in C. Actually many C programmers have been using these fundamental patterns in some form or another for years, often without clearly realizing it [10]. However, one shall deal with some tricky rules and techniques to implement OOP features in C such as embedding function pointers within the structs [7] and void pointers.

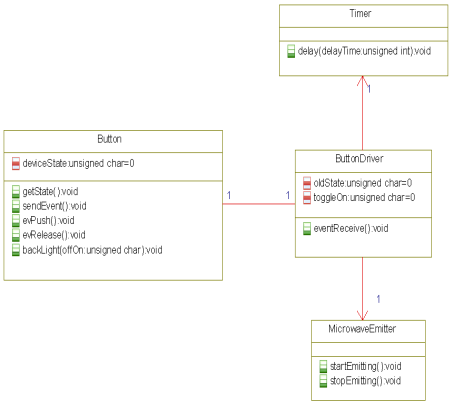


Figure 1. Debounce Example Model [7]

As a result of the experiments, Debounce example occupied 3628 16 bit – Words (Word) and Queue/CachedQueue example occupied 32294 Word total memory.

Table 1. Memory Allocation for the implementations

OOP Implementation	Memory Allocation (Word)	
	Debounce Example	Queue CachedQueue Example
In C	3628	32294
In C++	3692	31574

The same designs are implemented in an object-oriented language, C++. The code is intentionally written as simple as possible. These implementations are occupied 3692 Word and 31574 Word total memory, respectively.

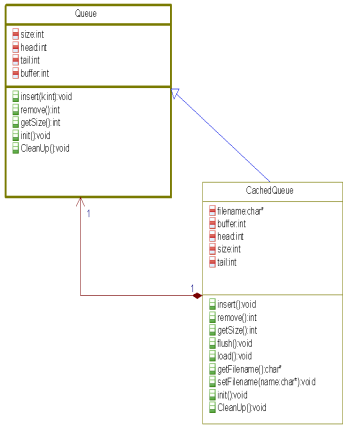


Figure 2. Queue/CachedQueue Example Model [7]

Note that, implementations were designed and coded carefully to make the comparison is fair. Complete results of the implementations are shown in Table 1.

In the second part of the experiment, the simple test program that simulates a single button press is run for Debounce example implementations. Then, the execution times of the implementations are recorded as Table 2.

Table 2. Execution Time for Debounce Example

OOP Implementation	Execution Time	
	Clock Cycle	*Time (µsec)*
In C	353	0,706
In C++	372	0,744

In addition, another simple test program [7] that shows elements inserted and removed into and from the queue is run for Queue/CachedQueue Example implementations. Then, the execution times of the implementations are recorded as Table 3.

Table 3. Execution Time for Queue CachedQueue Example

OOP Implementation	Execution Time	
	Clock Cycle	*Time (µsec)*
In C	251601	503,202
In C++	250922	501,844

Meanwhile, it is more meaningful not to ignore the effect of the compiler for the experiment. The performance of a DSP platform (DSP and compiler) depends upon the quality of the compiler [11]. Besides that, the performance of a compiler varies with the structure of the application and the programming style.

Table 4. Memory Allocation after the optimization

OOP Implementation	Memory Allocation (Word)	
	Debounce Example	Queue CachedQueue Example
In C	3244	31626
In C++	3276	31126

In the third part of the experiment, the above experiment is repeated by maximizing optimization levels of the compiler for each case to clarify the effect of the compiler to the experiment. Table 4 shows the memory allocation results for the implementations after optimization. In addition, Table 5 and Table 6 show the execution time results for the implementations after the optimization.

Table 5. Execution Time after the optimization for Debounce Example

OOP Implementation	Execution Time	
	Clock Cycle	*Time (µsec)*
In C	228	0,456
In C++	245	0,490

Table 6. Execution Time after the optimization for Queue/CachedQueue Example

OOP Implementation	Execution Time	
	Clock Cycle	*Time (µsec)*
In C	250339	500,678
In C++	250377	500,754

3. Discussion

Results of the experiment show that OOP implementation in C comes with almost same memory allocation, and execution time as OOP implementation in C++. With the above results, one might be convinced that OOP implementation in C shall be used in embedded systems because OOP in C does not have any overhead compared to OOP in C++. Actually, learning to use object-oriented techniques in C will not only make it easier to write your own objects in C, it will make it easier to understand the many toolkits and libraries that use these concepts [12]. However, Objects implemented in C are complex and the code becomes harder to debug and maintain. Even though, a tool is used to generate C code automatically from object-oriented design, implementations are hard to modify and maintain. The inclusion of object-oriented concepts into traditional languages sophisticated them, in that programmers had the flexibility to use or not to use the object-oriented extensions and benefits [13]. Although these languages became more complex, those extensions enabled programmers who had considerable experience with those traditional procedure languages to explore incrementally the different concepts provided by the object-oriented paradigm [13]. Nevertheless, when using a procedural language in OOP such as C, programmers had to exercise more discipline than when using a pure object-oriented language because it was too easy to deviate from sound object-oriented principles [13]. A powerful feature of object-oriented languages is the inheritance that allows classes to be arranged in a hierarchy and inherit behavior from classes above them. However, the danger in trying to force object-oriented concepts into a language that does not provide inheritance is that weird constructions may be produced, impairing software development, and jeopardizing the quality of the resulting software [13]. Specially, the polymorphism results in more errors and OOP is more difficult to recognize and understand, but again if a procedural language is used with an object-oriented design [15]. Actually, in embedded systems reliability is essential; indeed, embedded software may control a safety- or security-critical system where an error can have catastrophic consequences [8]. In addition, complexity is one of the important attribute of reliability and higher complexities increase the probability of error occurrences and decreases reliability of the software [17].

Alongside the recognized advantages, there seems to be a general feeling among procedural language programmers that object-oriented languages can result in inefficient code when compared with coding the same application in a procedural language. Like all such general knowledge, this need not be true; it all depends on which object-oriented language features you use and how you use them [14]. There are advantages of using an object-oriented language [16]. Object-oriented languages are able to distinguish an object's internal, add user-defined types to augment the native types, create new types by importing or reusing the description of existing types and localizes responsibility for behavior. In addition, the change of approach that comes with object orientation provides improved debugging and maintenance.

Meanwhile, C++ offers the embedded programmer some striking advantages over C [16]. It can be used in place of C without change for most applications. Nevertheless, most running C code compiles and runs as C++ code. It extends C by including additional critical features that support object-oriented and generic programming. C++ also embraced generic programming using templates. It remedies some of C's defects such as relying on the preprocessor, lack of type-safety, unrestricted casting. C++ also provides more scoping constructs and allows namespace scope and nested class scope, both unavailable in C.

As a point, C++ is a superset of C. This also demonstrates that moving to C++ is not an all or nothing event. Actually, the C programmer is nearly a C++ programmer. Moving from C to C++ is relatively simple and does not require a break with existing C practice [16]. It is also possible to choose among the C++ features those that are useful in the application and ignore others at the beginning.

Therefore, in complex software, the use of OOP in an object-oriented language such as C++ will lead to cleaner architecture, a better reuse of code, and one will start to get comparable timings or even a significant gain over a code with written in C.

Specially, real-time embedded applications require the features for promoting reliability, maintainability, reusability, and other broad software engineering goals such as compile-time type checking, support for encapsulation and information hiding, namespace management parameterizable templates and object-oriented programming features [8].

In addition that the supporting evidence was found that programmers produce more maintainable code with an object-oriented language as C++ than a standard procedural language as C [15].

However, DSP manufactures announce new generation chips, almost every year, offering improved performance, reduced code size and more on-chip memory to help the developers to implement their embedded systems in object-oriented languages with less memory and performance overheads.

Finally, I also keep in my mind that the effect of the compiler for my experiment. The results of third part of the experiment did not change the direction of previous results. Furthermore, I believe that effect of the compiler do not change the conclusion. However, surprisingly I have had almost the same optimization ratios for the implementation in C and C++ in terms of execution time and memory allocation contrary to common belief of C++ programs are harder to optimize than programs written in languages like C [15]. It also shows that the sophisticated compilers try hard to optimize the performance of object-oriented languages. Optimization percentages of the compiler for the implementations are shown in Table 7 and Table 8, respectively. Note that, Debounce example has higher optimization ratios in terms of memory allocation and execution time because it has short code length and the simpler test program compared to Queue/CachedQueue example.

Table 7. Optimization Ratios for Debounce Example

OOP Implementation	Optimization Execution Time	Optimization Total Memory
In C	% 35,41	% 10,58
In C++	% 34,13	% 11,26

Table 8. Optimization Ratios for Queue/CachedQueue Example

OOP Implementation	Optimization Execution Time	Optimization Total Memory
In C	% 0,50	% 2,06
In C++	% 0,21	% 1,41

Future work would deal with other procedural languages in order to generalize the discussion to OOP in procedural languages. In addition, future work would deal with power consumption as it is done with memory allocation and execution time.

4. Conclusion

According to the recent studies, it has been shown that OOP is a good methodology in embedded system research and development. ANSI C still is the most commonly used language for embedded system programming. Besides that, most of the features of OOP can be implemented via tricky techniques in C. However, OOP implementation in C adds the significant complexity to the implementations without any memory allocation and performance advantages. Nevertheless, the higher design and coding complexities increase the probability of error occurrences and decrease reliability of the software. Moreover, the new generation DSPs and compilers keep helping the developers to gain the advantages of object-oriented languages by improved performance, more on-chip memory, reduced and optimized code size. Consequently, there is no doubt that most new software systems will be object-oriented and will be implemented in object-oriented languages.

Acknowledgment

I would like to thank my colleagues and the anonymous reviewers for their helpful comments and suggestions.

References

[1] TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide: August 2006.

[2] M. Karjalainen, 1992. *Object-Oriented Programming of DSP processors: a case study of QuickC30, in International Conference on Acoustic, Speech and Signal Processing.* ICASSP-92 Vol 5, 1992

[3] A. Chatzigeorgiou, 2003. *Performance and power evaluation of C++ object-oriented programming in embedded processors, Information and Software Technology 45*, p. 195-201.

[4] M. Karjalainen, 1990. *DSP Software Integration by Object-Oriented Programming: a case study of QuickSig.* IEEE ASSP Magazine, 1990, p. 21-31.

[5] T. Afacan, 2011. *State Design Pattern Implementation of a DSP Processor: A Case Study of TMS5416C, Proceedings of the* IEEE Symposium on Industrial Embedded Systems (SIES), 2011, P.67-70.

[6] D. Batten, S. Jinturkar, J. Glossner, M. Schulte and P. D'Arcy, *A New Approach to DSP intrinsic Functions*. Proceedings of the 33rd Annual Hawaii International Conference on System Sciences, Jan. 2000, pp.908-918.

[7] B. P. Douglass, 2011. *Design Patterns For Embedded Systems in C.* Newness, Burlington, MA, USA,

[8] R. Philippe, H. Thomas, 2007. *Programming Embedded Systems, Seminar, Embedded Systems,* WS 06/07, Leopold-Franzens-University of Innsbruck Institute of Computer Science.

[9] Erh-Wen Hu, Cyril S. Ku, Andrew T. Russo, Bogong Su and Jian Wang, 2009. *Performance Analysis of Digital Signal Processors Using SMV*. Benchmark in International Journal of Signal Processing, 5;3, 2009, p. 223.

[10] M. Samek, 2002. P*ratical Statecharts in C/C++.* CMPBooks, San Francisco, CA, USA.

[11] M. Genutis, E. Kazanavicius, O. Olsen, 2001. *Benchmarking in DSP*, ISSN 1392-2114 ULTRAGARSAS, Nr.2 (39), 2001.

[12] G. Lebl, 2000. *Object Oriented Programming in C*, Linux Magazine, October 15th, 2000, http://www.linux-mag.com/id/628/.

[13] L. F. Capretz, 2003. *A Brief History of the Object-Oriented Approach,* ACM SIGSOFT, Software Engineering Notes. Vol. 28, No. 2, March, 2003 pp.1- 10.

[14] A. Lundgren, 2010. *The Inefficiency of C++, Fact or Fiction?,* EETimes Tech Papers, White Paper, June 2010.

[15] S. Henry, M. Humphrey, Science, Virginia Polytechnic Institute and State University, 1988. *Comparison of an Object-Oriented Programming Language to a Procedural Programming Language for Effectiveness in Program Maintenance,* Technical Report TR-88-49, Computer Science, Virginia Polytechnic Institute and State University, 1988.

[16] I. Pohl, 2001. C++ by Dissection, Addison Wesley, USA.

[17] A. Yadav, R. A. Khan, 2009. *Measuring Design Complexity – An Inherited Method Perspective,* ACM SIGSOFT, Software Engineering Notes. Vol. 34, No. 4, July 2009, pp.1- 5.

Dynamic Code Generation: an Experiment on Matrix Multiplication

Damien Couroussé Henri-Pierre Charles
CEA-LIST, Lastre laboratory
firstname.surname@cea.fr

Abstract

In this paper we detail the implementation of a typical CPU-bounded processing kernel: matrix multiplication. We used `deGoal`, a tool designed to build fast and portable binary code generators. We were able to outperform a traditional compiler: we obtained a speedup factor of 2.22 and 1.86, respectively for integer and floating-point multiplication with 256×256 matrices. Furthermore, code specialization on the data to process allows us to further increase the performance of the multiplication kernel by a factor of more than 20 in favorable conditions.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Translator writing systems and compiler generators; C.1.2 [*Processor Architectures*]: Multiple Data Stream Architectures—Parallel processors

General Terms Performance, Design, Algorithms

Keywords dynamic code generation, run-time optimization, embedded systems, parallel computing

1. Introduction

Since the early beginning of computer history, one has needed programming languages as an intermediary translation between algorithms and machine-readable instructions. Typically, from a simple viewpoint, running an algorithm on a computer requires the following steps: (1) the developer translates the algorithm into a source file containing programming language instructions, (2) a compiler translates these programming language instructions into machine code, (3) the processor reads and executes the machine instructions, loads the input data and produces the data results. Because compilation is performed *before* the program is run, it is not possible to produce machine code on the basis of knowledge of the execution context, which can be only known at run-time. This means that one has either to assume about the characteristics of the execution context (and to provide verification mechanisms), or to add extra instructions to adapt the program behavior. The other way to deal with this problem is to generate the program's machine code at run-time, after the execution context is known. This can be achieved by instruction translation or compilation at run-time [1]. A well-known example is the Java programming language, designed to enhance application portability: Java source code is written without a priori knowledge of the platform that will execute the final machine code, thanks to a virtual machine that does the match with the machine instructions supported by the target architecture.

Run-time compilation is also useful for large-scale parallel computer systems, where an application component can be populated on a lot of processing elements. This issue is applicable to all large-scale multi-processor platforms: from High Performance Computers in data centers to multiprocessor Systems-on-Chip (MPSoCs) in future embedded devices. In this case, one would need either (1) a generic implementation that one can parametrize at instantiation but that will suffer from the performance overhead brought by a generic implementation, or (2) to modify and re-compile the component dynamically at run-time after one knows where it will be finally executed.

`deGoal` was designed with the two issues described above in mind to provide application developers the ability to implement application kernels tunable at run-time depending on the execution context, on the characteristics on the target processor, and furthermore on the *data to process* [2]. In Just-In-Time compilers (JITs) *all* the application code is generated at run-time, which allows to perform optimizations covering the whole scope of the application, but also incurs a strong performance overhead. Usually in processing applications, most of the execution time is spent in a very small portion of the whole application source code, which is most of the time a computation-intensive task also called *kernel*. We assume that improving the performance of kernels can leverage the overall application performance. Therefore, the idea using `deGoal` is to embed *ad hoc* run-time code generators in a software application. Each code generator is specialized to produce the machine code of one application kernel. This enables the production of very fast code generators (10 to 100 times faster than common JITs).

The rest of this paper is organized as follows: section 2 introduces the core idea of `deGoal` and how this tool can be integrated in a larger-scale application, section 3 details the use of our tool on matrix multiplication for the processors of a MPSoC, section 4 details the results achieved, and section 5 presents related works.

2. Overview of `deGoal`

2.1 Kernels and compilettes

The two categories of software components around which our code generation technique is built are called *kernels* and *compilettes*:

Kernel A kernel is a small portion of code, which is part of a larger application, and which is most of the time under strong performance constraints; our technique focuses on the optimization at run-time of these small parts of a larger application in order to improve the kernel's performance. In the context of this paper, good performance is understood as low execution time and/or low memory footprint.

LCTES 2012 June 12–13, 2012, Beijing, China

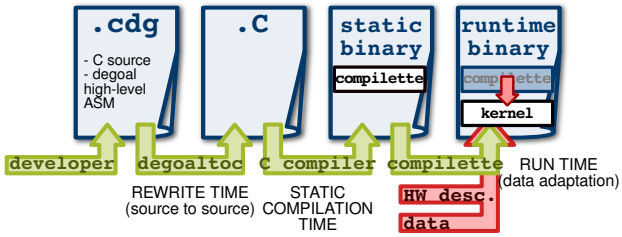


Figure 1. `deGoal` workflow: from the writing of application's source code to the execution of a kernel generated at run-time

Compilette A compilette is designed to generate the code of kernels at run-time. It can be understood as a small compiler that is executed at application's run-time. We use the term *compilette* to underline the fact in order to achieve very fast code generation, this small run-time compiler does not embed all the optimization techniques usually carried out by a static compiler. The binary code of a compilette is generated during the static compilation along with the rest of the application.

Compilettes are described using a mix of standard C and of a high-level ASM language [2], which describes the instructions that will be generated at run-time. However, on the contrary to common ASM languages, it is possible to parametrize these instructions with values known at run-time, and to use vector variables. More precisely, it is possible to manipulate vectors of registers, whose size will be determined at the time of code generation, when the use of registers in the programming context is known.

2.2 Workflow of code generation

The building of an application using `deGoal` is illustrated in figure 1 and explained below:

Writing the source code (application development time) This task is handled by the application developer, and/or by high-level tools. The source code of compilettes is written in specialized `.cdg` files, while the rest of the application software components are written using a standard programming language, such as C.

Generation of C source files (rewrite time) This step consists in a source-to-source transformation: the `.cdg` source files mixing high-level ASM instructions and standard C are translated into standard `C` source files by `degoaltoc`, which is one of `deGoal` tools. At this phase architecture-dependent features can be introduced in the C source files generated, for example register allocation and vectorization support.

Compilation of the application (static compilation time) The source code of the application now consists in a set of standard C source files, including the source code of the compilettes. The binary code of the application is produced by a standard C compiler. This step is the same as in the development of a standard C application.

Generation of kernel's binary code (run-time) At run-time, the compilette generates optimized binary code for the kernel(s) to optimize. This task can be executed on a processor that is different of the processor that will later run the kernel. Furthermore, the compilette's processor and the kernel's one do not necessarily need to have the same architecture. A compilette can be run several times, for example as soon as the kernel needs to be regenerated for new data to process. We have detailed on figure 1 two particular inputs of the compilette: data and hardware description. The originality of our approach indeed relies in the generation of a binary code optimized for a particular set of application data. At the same time, the code generation is able to introduce hardware-specific features.

```
clear(C)
for (y=0; y < n; y++) {
    for (x=0; x < q; x++) {
        for (i=0; i < p; i++) {
            C[x,y] = C[x,y] + A[i,y] * B[x,i]
        }
    }
}
```

Figure 2. Reference implementation of the matrix multiplication (in pseudo C code)

```
/* generation of the kernel's code */
(kernel, v) = compilette(A, B, C)

/* compute matrix multiplication */
clear(C)
for (y=0; y < n; y++) {
    for (i=0; i < p; i+= v) {
        kernel(y, i)
    }
}
```

Figure 3. optimized implementation of the matrix multiplication using `deGoal` (in pseudo-code)

Kernel execution (run-time) The program memory buffer filled by the compilette is run on the target processor (not shown in figure 1).

3. Implementation of matrix multiplication

This section describes the implementation of a processing kernel for matrix multiplication in order to illustrate the use of `deGoal`. We describe first a reference implementation, which is statically compiled with the platform's compiler. We then describe two improved implementations using `deGoal`: the first exploits matrix properties such as matrix size, element size, and memory addresses; the second exploits the values of matrix elements.

3.1 Reference implementation

Our aim is to perform matrix multiplication as described in equation 1, where a, b and c stand respectively for elements of matrices $[A]$, $[B]$ and $[C]$ of sizes $n \times p$, $p \times q$ and $n \times q$:

$$\forall i \in \{1,\dots,n\}, \forall j \in \{1,\dots,q\}, c_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj} \qquad (1)$$

The reference implementation of this algorithm is illustrated in figure 2. We used it as a reference implementation for our experimental measurements.

3.2 First implementation in a compilette

A simplified overview of our implementation of the matrix multiplication using `deGoal` is illustrated figure 3. `compilette` is the code generator that produces an optimized kernel function `kernel`, which encompasses the inner-most loop from figure 2: it performs a vector multiplication between a row in `A` and a column in `B`, and accumulates the result into the corresponding element of `C`. The code generated for `kernel` depends on the properties of matrices `A`, `B` and `C` : row and column sizes, memory alignment and address of the data in memory. These values are precomputed and propagated into the instructions of `kernel` at code generation time. In consequence, the only parameters needed by `kernel` are the row and column numbers of matrix `C`.

```
clear(C)

// generate the kernel's structure
(kernel_templ, v) = template_gen(A, B, C);

// process matrix multiplication
for (y=0; y < n; y++){
  for (i=0; i < p; i+=v){
    // specialize instructions on matrices' data
    kernel = data_gen(kernel_templ, A, y, i);
    if (NULL != kernel)
      kernel(y, i);
} }
```

Figure 4. Implementation of the matrix multiplication (pseudo-code) with code specialization on matrix values

This implementation of `kernel` is very similar to the reference implementation introduced above, at the exception that

- all the constants describing matrix properties, which are known at code generation time, have been propagated into the generated code.
- loops are reordered to minimize the number of memory loads. Considering the reference implementation of figure 2, we rearranged the loops to minimize memory loads for matrix `A`: the loop on `x` in done internally in `kernel`, and that the loop on `i` is raised one level up (figure 3). In other words, this means that once a line in matrix `A` is loaded, we compute all the related elements in matrix `C`.

As we will show in the results section, these improvements alone already contribute to a good improve performance.

3.3 Kernel specialization on matrix values

If the matrices to process are sparse or contain remarkable data values, it is possible to further increase performance by specializing the generated code depending on the element *values* of the matrix to process (figure 4). This time, the code generation is split in two phases: `template_gen` generates the global structure of the processing kernel that is not likely to change upon data values in A. At each processing loop, `data_gen` fills the kernel's code upon data values in the row vector to process in A. When there is nothing to execute (for example, all matrix values in the current row in A are null), `data_gen` returns `NULL` and we immediately move to the next loop step.

This technique involves an extra overhead for code generation because the kernel's code at each step in the innermost loop, but, as we will show below, this overhead can be compensated very quickly.

4. Experimental results

4.1 Target architecture

We target in this work the embedded platform called Platform 2012 (P2012) [6], under development by STMicroelectronics and CEA. It is composed of multiple clusters connected through an asynchronous network-on-chip allowing each cluster to have its own voltage and frequency domain. Each cluster aggregates 16 cores dedicated to processing, plus one extra core dedicated to task management. All of the cluster processors are STxP70-4 cores from STMicroelectronics.

We have added support for the STxP70 to `deGoal`. The STxP70-4 processor is a 32-bit RISC core. It comes with a variable-length instruction encoding and a dual VLIW architecture allowing two instructions to be issued and executed at each cycle. Two sets of hardware loop counters are provided to enable loop execution at maximum speed without cycle overheads due to software control. The core processor contains an internal extension for integer multiplication, and an optional single-precision floating point extension used in this experiment.

The P2012 SDK is delivered with a full toolchain for compiling, debugging, profiling and simulation in functional and cycle-accurate modes. Our experiment is based on the platform's toolchain and on the cycle-accurate simulator of the STxP70 core.

4.2 Experimental setup

We have evaluated our optimized version of the matrix multiplication against the reference implementation described in section 3.1.

The reference implementation is compiled in `-O3`. Loop unrolling, support of hardware loop counters and of the floating-point extension are also enabled. The best performance was obtained with an implementation close to the pseudo code described in figure 2.

The code generated by `deGoal`'s compilette does not depend on compiler optimizations, because it is generated at run-time by the compilette. Hence whatever the compiler optimizations selected, the execution time of the generated kernel remains constant. Compiler optimizations have however an effect on the performance of the compilette, because it is statically compiled as a standard application component. In our performance measurements, we have used the same compiler options to compare the reference implementation and our implementation using `deGoal`.

We have also exploited the VLIW extension of the STxP70-v4 core, using the appropriate compilation flags. On the compilette's side, VLIW support is integrated in the `cdg` pseudo-ASM language of `deGoal`. As a consequence, it is not exposed to the developer and the compilette is tailored to automatically exploit this feature as soon as the processor supports it.

4.3 Measure of the code generation time

We have instrumented the compilette to measure the time spent in code generation at run-time: code generation takes from 25 to 80 cycles per instruction generated. The speed of code generation varies significantly, mainly because of instruction bundling, and because of the extra computations done at the end of code generation, for example computing the jump addresses. The best results are achieved for unrolled loops without instruction bundling.

The code generation time is not taken into account in the speedup results presented below, because it is not necessary to regenerate the code for each matrix multiplication. As an indicator, code generation represents 15 to 20 % of the execution time for a multiplication of 16×16 matrices, and less than 0,01 % for 256×256 matrices.

4.4 Performance of the processing kernels

Figure 5 illustrates the performance improvements achieved using `deGoal` as compared to the reference implementation compiled with full optimization, for two cases of code generation: using the hardware loop counters provided by the STxP70 core (`HW loop`), and fully unrolling the kernel's code (`unrolled`). The speedup factor s represents the reduction factor of the execution duration of our implementation as compared to the reference implementation. We calculate it as follows: $s = \frac{t(\text{ref})}{t(\text{degoal})}$, where $t(\text{ref})$ measures the time execution of the reference implementation, $t(\text{degoal})$ the time execution of the generated kernel.

Our compilette brings a good overall performance improvement: when the matrix size is 256×256 elements, we achieve a reduction of the execution time of 2.22 times for integer multiplication, and of 1.86 times for floating-point multiplication.

Figure 6 illustrates the speedup factor measured when using code specialization on the data of matrix A, as presented in section 3.3. We illustrate here the most favorable case where matrix A is the identity matrix. In this case, the looped implementation shows a huge speedup because of the instructions removed from the kernel

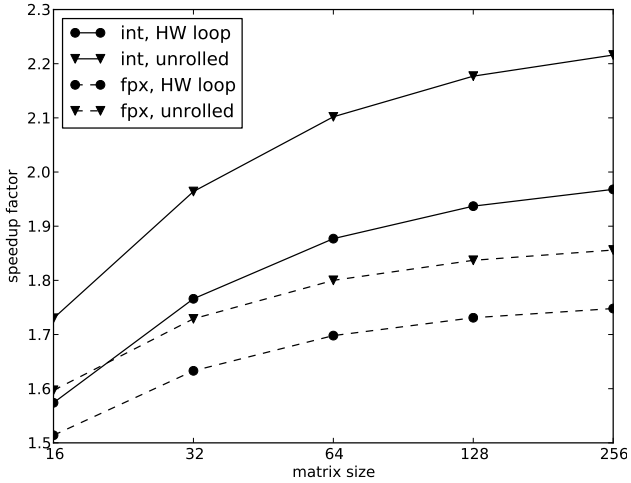


Figure 5. Speedup factor measured, for integer multiplication (plain line) and floating-point multiplication (dashed line), according to the implementation described in section 3.2.

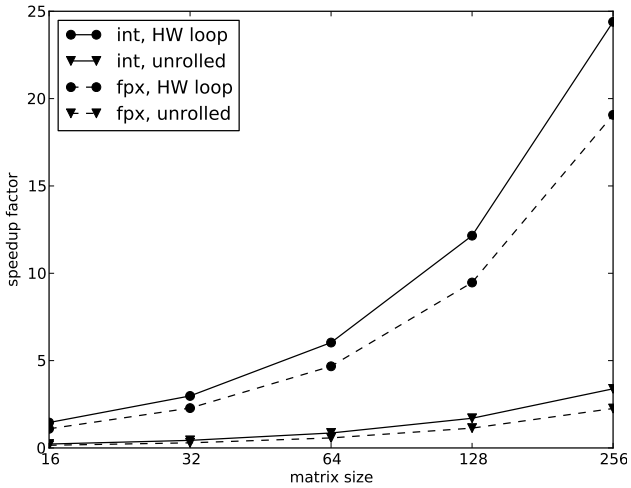


Figure 6. Speedup factor measured, for integer multiplication (plain line) and floating-point multiplication (dashed line), according to the implementation described in section 3.3.

when null values are met in matrix A. The unrolled version is not efficient, considering the favorable experimental conditions, because a part of the code generation is performed *during* kernel's execution, and code unrolling requires a lot more instructions to be generated.

5. Related work

There is an extensive amount of literature about dynamic compilation, mainly related to Just-In-Time compilers (JITs) [1]. JITs dynamically select the parts of the program to optimize without a priori knowledge on the input code. This usually requires to embed a large amount of intelligence in the JIT framework, which means a large footprint and a significant performance overhead. In order to target embedded systems, some research works have tried to tackle these limitations: memory footprint can be reduced to a few hundreds of KB [4], but the binary code produced often presents a lower performance because of the smaller amount of optimizing intelligence embedded in the JIT compiler [5].

The approach chosen in `deGoal` is similar to partial evaluation techniques [3], which consists in pre-computing during the static compilation passes the maximum of the generated code to reduce the run-time overhead. At run-time, the finalization of the machine code consists in: selecting code templates, filling pre-compiled binary code with data values and jump addresses. Using `deGoal` we compile statically an *ad hoc* code generator for each kernel to specialize. The originality of our approach relies in the possibility to perform run-time instruction selection depending on the *data* to process [2].

Our approach allows to generate code at least 10 times faster than traditional JITs: JITs hardly go below 1000 cycles per instruction generated while we obtain 25 to 80 cycles per instruction generated on the STxP70 processor.

6. Conclusion

We have shown that `deGoal` can easily compete with a highly optimized code produced by a static compiler with little effort: the code produced has better performance than a code statically compiled with full optimization, and furthermore the quality of the code produced with `deGoal` is consistent and does not depend on compiler's options. `deGoal` also allows to specialize the code of a processing kernel for a particular set of run-time data, which is not possible using a static compiler. We have shown that in favorable conditions the performance increase can be huge.

In this paper, we have illustrated the benefits of using `deGoal` to optimize processing kernels. Because `deGoal` is related to the generation of machine binary instructions, its scope is actually restricted to the processor. In order to use these optimization techniques in large scale platforms, e.g. MPSoCs or HPC clusters, one must rely on tools of higher level for the parallelization of an application on multiple processing elements. Future work will present how it is possible to integrate kernels optimized with degoal's compilettes in large scale applications.

`deGoal` is currently under active development. It is able to produce code for multiple platforms: Nvidia GPUs, ARM processors, the STxP70, and other RISC processors under NDA.

Acknowledgments

The authors wish to acknowledge the support of the EU Commission under the SMECY project (ARTEMIS Joint Undertaking under grant agreement number 100230) in part funding the work reported in this paper.

References

[1] J. Aycock. A brief history of just-in-time. *ACM Computing Surveys*, 35: 97–113, June 2003.

[2] H.-P. Charles. Basic infrastructure for dynamic code generation. In H.-P. Charles, P. Clauss, and F. Pétrot, editors, *workshop "Dynamic Compilation Everywhere", in conjunction with the 7th HiPEAC conference*, Paris, France, january 2012.

[3] C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In *Proceedings of the 23th Annual Symposium on Principles of Programming Languages*, pages 145–156, 1996.

[4] A. Gal, C. W. Probst, and M. Franz. HotpathVM: an effective JIT compiler for resource-constrained devices. In *VEE '06*, pages 144–153, New York, NY, USA, 2006. ACM.

[5] N. Shaylor. A just-in-time compiler for memory-constrained low-power devices. In *Java VM'02*, pages 119–126, Berkeley, CA, USA, 2002. USENIX Association.

[6] STMicroelectronics and CEA. Platform 2012: A many-core programmable accelerator for ultra-efficient embedded computing in nanometer technology. In *CMC Research Workshop on STMicroelectronics Platform 2012*, 2010.

The Static Analysis of Worst-Case Inter-Core Communication Latency in CMPs with 2D-Mesh NoC

Yiqiang Ding Wei Zhang
Electrical and Computer Engineering,Virginia Commonwealth University
{dingy4,wzhang4}@vcu.edu

Abstract

Network-on-Chip (NoC) is adopted to provide fast and efficient communications in chip multiprocessors (CMPs), especially for many-core processors. However, dynamic processor allocation and job scheduling in CMPs make it hard to predict the traffic patterns in NoC statically, therefore it is complex and challenging to analyze the worst-case latency of the communications from a real-time application executing on CMPs with NoC, which is important to obtain the worst-case execution time (WCET) of the application. In this paper, we study the static analysis of the maximum value of the worst-case latencies of all possible communications in a CMP with a packet-switching 2D-Mesh NoC, which is called the worst-case inter-core communication latency (WICL). A basic approach is proposed to estimate the WICL of a 2D-Mesh NoC in the ideally worst-case scenario. Our experiments show that the overestimation is within 80%.

1. Introduction

Chip multiprocessors (CMPs) have become an attractive approach to build high-performance multi-core real-time systems. In CMPs, delays caused by wires dominate over those generated by gates, which favors short and energy efficient links rather than long buses. Therefore Network-on-chip (NoC) has become the best approach to provide fast and efficient communications in CMPs, especially for many-core systems.

The latency of the communication in NoC is a part of the execution time of an application executing on CMPs with NoC. Also as CMPs are used in the hard real-time systems, it is desirable to include the worst-case latency of communications in NoC into the analysis of the worst-case execution time of a real-time application. However, it is quite challenging to analyze the worst-case latency of the communication in NoC for a real-time application running in CMPs with NoC because of the following reasons: First, the traffic characteristics of the target application can not be exactly known before run-time. To be specific, as dynamic processor allocation is usually adopted in CMPs, the core assigned to the target application is determined at run-time. So the source location in NoC of the traffic from the target application could be any node in the NoC. Also as the traffics from the target application are either between the core and the memory or between two cores, both the destination location and the sending speed of the traffic from the target application are hardly known before the run-time; Second, the worst-case latency of traffic from a target application not only includes the latency to be transmitted between the source node and the destination node, but also includes the delay caused by the contention from other traffics of co-running applications in NoC. However, both dynamic processor allocation and job scheduling can lead to various traffic patterns for possible co-running traffics, which are hardly predicted statically. Furthermore, some techniques used in NoC to improve the average network performance, for example adaptive routing algorithms, make the analysis of the worst-case latency of the communications of a real-time application executed in CMPs with NoC, if not possible, quite complicated.

There are some research works aiming at providing guaranteed timing requirements for NoCs. Multiple techniques are proposed in these works, including the support of special hardware mechanisms [4], using priority-based mechanisms [10], time-triggered systems [7], AEthereal network [2], and time division multiple access [9]. Also some researchers have studied the analysis of the inter-core communication latency in NoCs. Fadi Sibai [11] calculated the inter-core communication latency in NoCs with different types of topology by accumulating the average latency to pass each router on an inter-core communication path. S. Foroutan, et.al.[6] proposed to construct a reduced Markov chain model for each node of the inter-core communication path and recursively use the local mean latencies to obtain the mean latency of the complete path. However, these works only consider the average-case inter-core communication latency in NoCs. Furthermore, T. Ferrandiz et.al.[5] proposed a method to compute an upper-bound on the worst-case inter-core communication delay of a packet in a SpaceWire network (a special type of network-on-chip). It assumes there is no queue (buffer) in the input links and estimates the upper-bound of the latency to transfer a packet through a link depending on the worst-case delay to wait for other packets to be transferred through this link before it. Y. Qian et. al [8] presented an analysis technique to derive per-flow communication delay bound. Based on a network contention model, this technique employs network calculus to first compute the equivalent service curve for an individual flow and then calculated its packet delay bound. However, all these works are done based on the assumption that the traffic patterns in a given NoC (like the number of flows, the speed, the source and destination of each flow) are known a priori, and the worst-case inter-core communication latency is calculated for each known flow.

The static analysis of the WICL of a 2D-Mesh NoC can help support the static WCET analysis of the real-time application executing on a CMP which inter-core communications are based on the NoC, where dynamic processor allocation and job scheduling lead to the uncertainty of the traffic patterns in the NoC before run-time; A basic approach is proposed to estimate the WICL of a homoge-

neous 2D-Mesh NoC in case of the ideally worst-case scenario; it first bounds the worst-case waiting delay in each router on the path of a traffic-flow from the worst-case traffic pattern of the ideally worst-case scenario, which also defines the *worst-case input channel contention* and the *worst-case output channel contention* in a router; then the worst-case latencies of all traffic-flows are calculated and sorted to find the maximum value which is considered as the observed WICL of the NoC. And the traffic-flow with the WICL is named as the worst-case traffic-flow of the 2D-Mesh NoC.

2. The Worst-case Inter-Core Communication Latency

In our study, the topology of the NoC in CMPs is assumed to be 2D-Mesh as shown in Figure 1. The NoC consists of multiple routers, each of which is attached by a core with a link, and two adjacent routers in the same row/column are connected by two directed links. As shown in Figure 2, the core component of a router is the crossbar switch, and in general there are one input channel and one output channel connected with one side of the crossbar switch. For convenience, the directions of four sides of the crossbar switch are named as North, East, South and West in clockwise order. Also each input/output channel on each side is connected with the input/output link attached to the router from the same direction. The NoC studied is based on packet switching mechanism [3], so each input channel has a buffer to queue the incoming packets from the input link connected with it, while each output channel only has the space for a packet transmitted currently on the output link connected. In this paper, we assume the store-and-forward flow control [12] is used in the router using packet switching, and the packets queued in the buffer of an input channel are scheduled by the fifo scheduling algorithm.

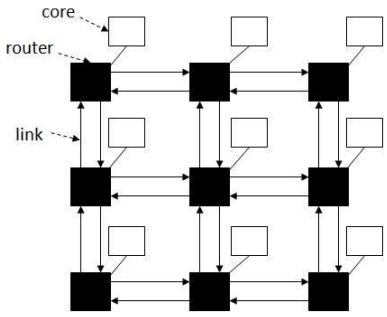


Figure 1. The architecture of the 2D-Mesh NoC in CMPs.

As the latencies of the inter-core communications are an important part of the execution time of a real-time application running on CMPs with a 2D-Mesh NoC, it is desirable to include the analysis of the worst-case latency of the inter-core communication into the static WCET analysis of the real-time application. However, The static analysis of the WICL in a 2D-Mesh NoC is challenging because of the uncertainty of traffic characteristics in the NoC caused by the dynamic processor allocation and job scheduling applied in CMPs. First, the source and the destionation of the inter-core communications of the real-time application analyzed is hard to know statically, because it can be allocated to any core in the CMP at run-time even though job migration among the cores is not considered, also it possibly communicates with any application allocated to other cores at run-time; Second, the traffic characteristics of the inter-core communications from other co-running applications before run-time as well.

An inter-core communication can be described as a *traffic-flow* which consists of multiple packets traversing the NoC at the same route, and these packets are sent from the source core at a fixed/various speed. For a packet, the time during the transmission between its source and destinatin is denoted as the *packet network latency*. In a 2D-Mesh NoC, given the deterministic routing algorithm, the link bandwidth and the packet size, assuming the route of a traffic-flow F includes a set of routers Set_r and a set of links Set_l, the packet network latency of a packet P in F can be represented by Equation 1 which includes the sum of the transmit latencies on all routers in the route and the sum of the transmit latencies of all links as well. Because the packet size and the link bandwidth are both fixed, the transmit latencis on the links for all packets in a traffic-flow are the same, and the variation of the packet network latency of different packets in a traffic-flow originates from the variation of the transmit latency in the routers which can be represented by Equation 2. While the latency of routing ($T_{routing}$) and the latency to pass the crossbar switch (T_{switch}) are both fixed given a specific configuration of the router, the waiting delay in the router (T_{wait}) not only depends on the configuration of the router, such as the flow control mechanism, the scheduling algorithm and the buffer size, but also is affected by the traffic characteristics of other concurrent traffic-flows with F, because the resources in a router are contended by the packets from all possible traffic-flows passing it.

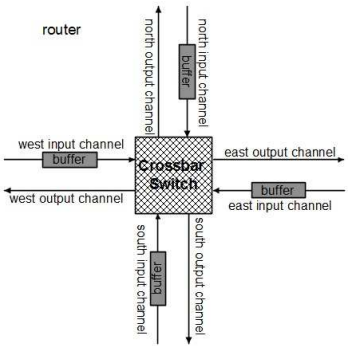


Figure 2. The architecture of the router in a 2D-Mesh NoC.

$$Network\ Latency\ of\ P = \sum_{Router_i \in Set_r} latency\ in\ Router_i + \sum_{Link_j \in Set_l} latency\ on\ Link_j \quad (1)$$

$$Latency\ in\ R_i\ for\ P = T_{routing} + T_{switch} + T_{wait} \quad (2)$$

Although the packet network latencies of various packets in a traffic-flow vary because of the various waiting delay in each router at the route, the maximum of all packet network latencies of the traffic-flow should be bounded and denoted as *the worst-case latency of a traffic-flow*. Thus if an inter-core communication of an application can be represented by this traffc-flow, the total latency of the inter-core communication in the worst-case can be estimated by the worst-case latency of this traffic-flow and the number of packets to be transmitted. It should be noted that the estimation of the number of packets to be transmitted is out of the scope of this study. However the uncertainty of the processor allocation for an application executing on a CMP with a 2D-Mesh NoC before run-time makes it difficult to know the traffic-flow representing its inter-core communication statically, which could

happen between any two cores. It is unsafe to use the worst-case latency of any traffic-flow to estimate the total latency of the inter-core communication of an application in the worst case. Hence it is necessay to bound the maximum value of the worst-case latency of any possible traffic-flow in a 2D-Mesh NoC, namely the worst-case inter-core communication latency (WICL) of the 2D-Mesh NoC.

One the one hand, the foundation of the static analysis of the WICL of a 2D-Mesh NoC is to explore the worst-case scenario where the worst-case network resource contentions happen under the possible traffic patterns in the NoC (eg. the overall traffic characteristics of the concurrent traffic-flows in the NoC). On the other hand, it should consider the effects from the configuration of the routers in the NoC, such as the flow control, the scheduling algorithm, the routing algorithm, and the size of the buffer. It also assumes that there is no packet loss during the transmission, because if any packet is lost, the worst-case latency of the traffic-flow which the packet is in should be considered as infinite, so the WICL must be infinite (or can not be bounded), which does not make sense for the static WCET analysis of a real-time application. The 2D-Mesh NoC studied in this paper uses two routing algorithms respectively: X-Y routing [12] and Odd-Even (OE) routing [12], and store-and-forward flow control with fifo scheduling is used.

3. Ideally Worst-Case Scenario

Assuming an $N \times N$ 2D-Mesh NoC with N^2 cores, it is possible that there exist the maximum $N^2 \times (N^2 - 1)$ traffic-flows simultaneously in the NoC, in case that each core is multicasting $N^2 - 1$ traffic-flows to other $N^2 - 1$ cores respectively. It is called as the *ideally worst-case traffic pattern*. The path of a traffic-flow under a deterministic routing algorithm is fixed, and it includes multiple routers and the links connected with them. Given a number of traffic-flows, the number of traffic-flows passing a link, the input channel and the output channel connected with the link can be calculated and is called as the *traffic-flow weight* of these entities. In the ideally worst-case traffic pattern, the number of traffic-flows passing an entity reaches the maximum value which is called as the *worst-case traffic-flow weight of* this entity.

As mentioned in Section 2, the packet network latencies of the packets in a traffic-flow vary because of the *waiting delay* in the router. Therefore the worst-case latency of a traffic-flow happens with the worst-case waiting delay in each router on its path in case of the worst-case traffic pattern. The worst-case waiting delay of a packet P from a traffic-flow F in a router R happens if the worst-case contention happens when this packet passes the router. The worst-case contentions can be classfied into two aspects as follows:

1. **The worst-case input channel contention**: if P enters R from the input channel C_{input}, and C_{input} is in the state of the worst-case traffic-flow weight W_{input}, which means there are W_{input} packets including P in the buffer of C_{input} from all traffic flows passing C_{input}, P is transmitted after the transmission of all other W_{input}-1 packets;
2. **The worst-case output channel contention**: if P exits R from the output channel C_{output}, and C_{output} is in the state of the worst-case traffic-flow weight W_{output}, which means there are W_{output} packets including P requiring the transmission from C_{output} from all traffic flows, P is transmitted after the transmission of all other W_{output}-1 packets;

Besides the worst-case contentions, the estimation of the worst-case waiting delay for a packet from a traffic-flow differs by using different flow control mechanisms and packet scheduling algorithms. In case of store-and-forward flow control and fifo scheduling, the worst-case waiting delay for a packet P from a traffic-flow F in a router on its path can be estimated following Algorithm 1 which satisfies the worst-case contentions in both the input channel and the output-channel. The variables used in the algorithm are explained from Line 1 to 10. As the packets from the same input channel IC_t with P can exit the router either from OC_t or other output channels (except the output channel in the same direction with IC_t), the worst-case waiting delay of P is calculated by checking all output channels which passed by any packet from IC_t. For any output channel OC_j passed by the packets from IC_t, the worst-case delay for P to wait for the transmission of the packets from other input channels can be represented by $(N_j - N_{t\_j}) \times T_l$ as shown in Line 15 and 17 according to the worst-case output channel contention model. In general, the worst-case delay to wait for the transmission of the packets from IC_t to OC_j equals to $(T_s + T_r + T_l) \times N_{t\_j}$ as shown in Line 17 according to the worst-case input channel contention model. Especially in case that OC_j is OC_t, the number of packets to wait for by P is $N_{t\_j} - 1$ as shown in Line 15 because P is also counted in N_{t_j}.

Algorithm 1 Worst-case Waiting Delay Analysis

```
1:  D_w: the worst-case waiting delay of P in the router
2:  IC_t: the input channel where P enters the router
3:  IC_i: any input channel of the router
4:  OC_t: the output channel where P exits the router
5:  OC_j: any output channel of the router
6:  T_s: switching latency of a packet in a router
7:  T_r: routing latency of a packet in a router
8:  T_l: transmit latency of a packet through the link connected with OC_j
9:  N_{t_j}: the number of flows exiting the router through OC_j from IC_t
10: N_j: the total number of flows exiting the router through OC_j
11: begin
12: for each OC_j except the output channel in the same direction with IC_t do
13:     if N_{t_j} >0 then
14:         if OC_j is OC_t then
15:             D_w += (T_s + T_r + T_l) × (N_{t_j} − 1) + (N_j − N_{t_j}) × T_l
16:         else
17:             D_w += (T_s + T_r + T_l) × N_{t_j} + (N_j − N_{t_j}) × T_l
18:         end if
19:     end if
20: end for
21: return D_w
22: end
```

By integrating the worst waiting delay of a packet of a traffic-flow in each router on its path into Equation 1 and Equation 2, the worst-case latency of the traffic-flow can be calculated. Therefore, the WICL of a 2D-Mesh NoC in the ideally worst-case scenario can be estimated by the basic approach as followings:

1. Calculate the path for each traffic-flow under the ideally worst-case traffic pattern according to a deterministic routing algorithm;
2. Calculate the worst-case traffic-weight of all input channels and output channels in each router of the NoC;
3. Calculate the worst-case latency of each traffic-flow under the ideally worst-case traffic pattern;
4. Sort the worst-case latencies of all the traffic-flows, and the maximum latency is considered as the WICL of the NoC.

4. Evaluation Methodology

In order to validate the basic approach, an analyzer is built to estimate the WICL of a 2D-Mesh NoC, and the NoC simulator Nirgam [1] is extended to support the simulation of the ideally worst-case scenario. In addition, the simulator adopts some intermediate results outputted from the analyzer,which is the buffer size of the input channel of the routers.

The latency is measured in CPU cycles; the packet size is set as 5 bytes and the bandwidths of all links are set as 5 bytes/cycle. The network size ranges from 2×2, 3×3, 4×4, 5×5, 8×8 to

	X-Y routing		
size	estimated	observed	estimated/observed
2×2	27	25	1.08
3×3	133	114	1.17
4×4	406	299	1.36
5×5	977	63	1.53
8×8	6067	3456	1.76
10×10	14464	8016	1.80

Table 1. the comparison of the WICL of a 2D-Mesh NoC using X-Y routing in both estimated scheme and observed scheme with different network sizes, which is measured in cycles

	OE routing		
size	estimated	observed	estimated/observed
2×2	31	28	1.11
3×3	239	198	1.21
4×4	918	654	1.40
5×5	3190	2048	1.56
8×8	33924	18256	1.85
10×10	122444	65472	1.87

Table 2. the comparison of the WICL of a 2D-Mesh NoC using OE routing in both estimated scheme and observed scheme with different network sizes, which is measured in cycles

	X-Y routing		OE routing	
size	estimated	observed	estimated	observed
2×2	3	3	3	3
3×3	5	5	7	7
4×4	7	7	10	10
5×5	9	9	17	17
8×8	15	15	37	37
10×10	19	19	55	55

Table 3. the number of hops in the worst-case traffic-flow in both estimated case and observed case with different network sizes

10×10. The router uses X-Y routing and OE routing respectively, and both routing and switching are assumed to cost 1 cycle. The size of the buffer in each input channel is set to the total size of multiple packets, which equals to the maximum of the worst-case traffic-weight of the input channels of a given 2D-Mesh NoC.

5. Experimental Results

As shown in Table 1 and Table 2, the estimated WICL of a 2D-Mesh NoC is larger than the observed one for each network size in the ideally worst-case scenario by using X-Y routing and OE routing respectively; As it is possible that the estimated worst-case traffic-flow is different from the observed one, in order to verify the worst-case traffic-flow estimated by the basic approach, Table 3 compares the number of hops in the worst-case traffic-flow between estimated and observed for each network size and each routing algorithm. The results demonstrate that the basic approach can safely bound the WICL of a 2D-Mesh NoC with both X-Y routing and OE routing in the ideally worst-case scenario.

However, the estimated WICLs from the basic approach are not accurate comparing with the observed ones. The overestimation mainly comes from the worst-case all-to-all traffic pattern, and strictly worst-case contention in a router assumed in the worst-case scenario. With the increase of the networ size, it is more difficult to achieve these two worst-case conditions in a 2D-Mesh NoC and each router in it in the simulation. So the overestimation from the estimated results increases if the network size is enlarged. In addition, the WICL as well as the overestimation in OE routing is larger than those in X-Y routing, because OE routing does not lead to the shortest path for a traffic-flow, but X-Y routing does.

6. Conclusions

Although NoC can provide fast and efficient inter-core communications to real-time systems, it brings new challenge to WCET analysis of real-time applications running on CMPs. It is desirable to statically obtain the worst-case latency of the inter-core communications of the applications, which is difficult because of the uncertainty of the traffic pattern before run-time brought by dynamic processor allocation and job scheduling in CMPs. In this paper, a basic approach is proposed to estimate the WICL of a 2D-Mesh NoC with two routing algorithms in the worst-case scenario, and it can estimate the WICL safely but not tightly according to the experimental results. In the future work, we plan to study an enhanced approach to estimate the WICL of a 2D-Mesh NoC more accurately in a realistically worst-case scenario. Also, we would like to explore the WICL analysis for applications with static task-to-core mappings.

References

[1] Nirgam: A simulator for noc interconnect routing and application modeling. http://nirgam.ecs.soton.ac.uk/home.php.

[2] K. Goossens, J. Dielissen, and A. Radulescu. AEthereal Network on Chip: Concepts, Architectures, and Implementations. In IEEE Design and Test of Computers. September-October, 2005.

[3] W. Dally and B. Towles. Route packets, not wires: on-chip interconnection networks. In DAC, 2001.

[4] J. Diemer and R. Ernst. Back suction: Service guarantees for latency-sensitive on-chip networks. In ACM/IEEE International Symposium on NOCS, 2010.

[5] T. Ferrandiz, F. Frances, and C. Fraboul. A method of computation for worst-case delay analysis on spacewire networks. In *Industrial Embedded Systems, 2009. SIES '09. IEEE International Symposium on*, pages 19 –27, july 2009.

[6] S. Foroutan, Y. Thonnart, R. Hersemeule, and A. Jerraya. A markov chain based method for noc end-to-end latency evaluation. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1 –8, april 2010.

[7] C. Paukovits and H. Kopetz. Concepts of switching in the time-triggered network-on-chip. In *Embedded and Real-Time Computing Systems and Applications, 2008. RTCSA '08. 14th IEEE International Conference on*, pages 120 –129, aug. 2008.

[8] Y. Qian, Z. Lu, and W. Dou. Analysis of worst-case delay bounds for on-chip packet-switching networks. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 29(5):802 –815, may 2010. ISSN 0278-0070.

[9] J. Rose, P. Eles, Z. Peng, and A. Andrei. Predictable worst-case execution time analysis for multiprocessor systems-on-chip. In *Electronic Design, Test and Application (DELTA), 2011 Sixth IEEE International Symposium on*, pages 99 –104, jan. 2011.

[10] Z. Shi and A. Burns. Real-time communication analysis for on-chip networks with wormhole switching. In *Proceedings of the Second ACM/IEEE International Symposium on Networks-on-Chip*, NOCS '08, 2008.

[11] F. Sibai. Resource sharing in networks-on-chip of large many-core embedded systems. In *Parallel Processing Workshops, 2009. ICPPW '09. International Conference on*, pages 513 –519, sept. 2009.

[12] B. T. William J. Dally. *Principles and practices of interconnection networks*. Morgan Kaufmann, 2004.

Acknowledgments

This work is partially supported by NSF grant CCF 0914543.

Adaptable and Precise Worst Case Execution Time Estimation Tool

Vladimir-Alexandru Paun
UEI, ENSTA ParisTech
Paris, France
paun@ensta-paristech.fr

Bruno Monsuez
UEI, ENSTA ParisTech
Paris, France
monsuez@ensta-paristech.fr

Abstract

Real-time systems are everywere. When they are integrated into safety-critical systems, the verification of their properties becomes a crucial part. Besides the growth in complexity of the embedded systems, platforms are getting more and more heterogeneous. Being able to validate their non-functional properties is a complex and resource consuming task. One of the main reasons is that currently available solutions focus on delivering precise estimation through tools that are highly dependent on the underlying platform as in order to provide precise and safe results, the architecture of the system must be take into account. In this project we address these issues by developing a prototype that maintains a good level of precision while being adaptable to a variety of platforms by separating as much as possible the worst case execution time estimation stage from the hardware modeling aspects.

General Terms Hard Real-Time Systems, precision, safety, adaptability

Keywords WCET, Abstract State Machine, Symbolic Execution

1. Introduction

With regard to the respect of the timing constraints, real-time systems are classified in two categories: hard real-time systems (the non respect of a deadline can lead to catastrophic consequences) and soft real-time systems (missing a deadline can cause performance degradation and material loss). We analyze hard real-time systems that need precise and safe determination of the worst case execution time (WCET) bounds that are crucial in the certification process. Traditionally two approaches are used, namely dynamic and static methods [1]. We only consider the latest as dynamic methods, in the traditional sense, fail to deliver safe estimations for modern platforms that contain, for example, pipelines or cache memories and tend to greatly underestimate the WCET.

In order to give a safe estimation of the WCET, all the interactions and reachable states of the system must be analyzed or over approximated, hence the need of an analysis that takes into account the exact underlying architecture. We choose to separate as much as possible the modeling part from the analysis part in order to achieve the flexibility needed to adapt to new hardware.

LCTES '12 12-13 June, Beijing.

In our approach we start from the system's model and the binary that will be executed on the final platform. An extension of the Symbolic Execution (SE) [2], the *conjoint* SE, will generate all the reachable states of the processor, under the supervision of a *prediction module* that will *fusion* identical and similar states in order to contain the state space explosion and give details regarding the global precision loss of the WCET estimation.

For the processor model we choose to use a model based on the Abstract State Machines (ASMs). The major advantages of using ASMs for the processor modelings can be summarized as follows: shortness of description (e.g. 200 lines for the ARM7 processor [3]), readability of the specification, cycle accuracy, acceptable simulation speed and the ease of conception (the ASM Refinement Method - piecemeal decomposition of a system into constituent parts which are treated separately to manage complexity - the ASM refinements can then be verified using generalized forward simulation for example [4]).What further differentiates the ASM model is the possibility to prove its correctness using several formal verification approaches (e.g. by model checking, [5] based on the ASM Workbench, [6], a comprehensive tool environment supporting the development and computer-aided analysis and validation of ASM models). Daho et all. use TLA+ logic for the deductive verification of ASMs in [7].

In the following we first take a look into the state of the art concerning timing analysis and we continue with the description of the high level architecture of our tool. Subsequently we take a closer look into the formal model used to simulate the hardware that gives us the edge in the adaptability of our tool followed by a presentation of the WCET estimation steps and the transformations needed to contain the combinatorial explosion.

2. Related works

Many of the available timing analysis tools show a list of compatible hardware and present each new platform taken into account as a new feature. OTAWA, introduced by Casse and Sainrat [8], is a toolbox designed to enable the implementation of research algorithms that are combined in order to compute estimations of the WCET. Their abstraction layer separates the analysis from the target hardware and the instruction set architecture. The use of the parametrized model of a generic platform helps thus addressing a variety of architectures. However, the model seams to lack precision as it fails to capture the precise behavior of the platform. AbsInt's `a^3` tool determines the WCET through several phases, as we can see in [9] and [10]. It uses abstract interpretation for the value analysis, the cache analysis, and the pipeline analysis (e.g. in building the set of possible processor states in input/output of each basic block). Each hardware analysis provides an abstract semantics of the hardware that describe the behavior of those components

on the abstract values. This step must be repeated for every new architecture taken into account.

3. The global architecture of the WCET estimation tool

The two main entries of the tool are the processor model and the program binary, as depicted in Figure 1. The processor is regarded as the union of its components $\mu P = \bigcup_{i=0}^{n} C_i$ and modeled as a hierarchical timed abstract state machine, described further in the paper, that has the useful feature of enabling multiple definitions for a same component C_i. A supervisor that we call the *Oracle* decides what abstraction level is best suited for the current context in order to optimize the *precision* to *state explosion* ratio. A value analysis stage is used to obtain information regarding the instruction order, their addresses and the control flow graph of the program. Symbolic execution is used to symbolically execute each instruction of the program. This means that each variable has initially a symbolic value (as we generally do not posses exact information on its value) that gets refined by accumulating all the informations and decisions taken during execution. One of the advantages of this method is that it manages to simulate the interactions inside the processor in detail, for example capturing by construction the timing anomalies [12]. The *SE* generates all reachable states of the processor, meaning that we have to manage a rapidly increasing state space. Our fusion stage consists in merging as much states as possible without affecting too much the precision of the estimation. We achieve this by using the prediction module that will first identify the states that are good candidates for merging and then estimate the impact of the fusion on the global analysis. After browsing and evaluating the processor's states, the time corresponding to the worst path is selected.

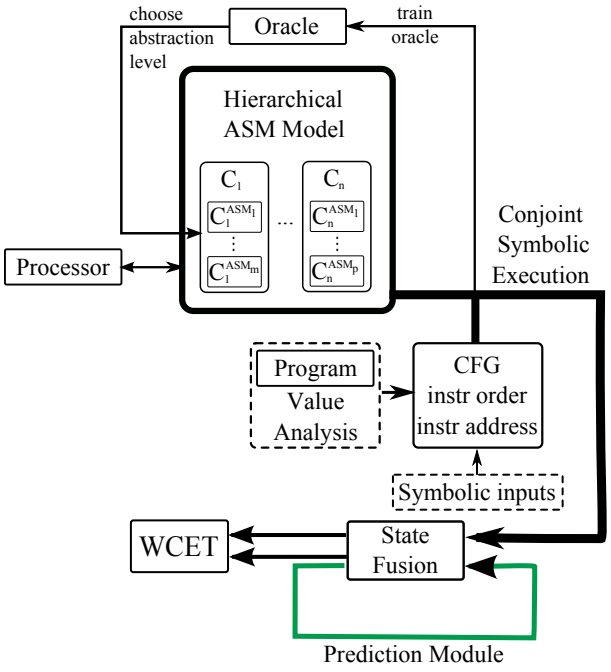


Figure 1. Global architecture of the WCET estimation tool

4. Timed Hierarchical Abstract State Machines

4.1 Abstract State Machine Formalism

The sequential ASM Thesis, introduced in [13] proves the isomorphic modeling of any algorithm. The sequential ASM algorithm consists of a set of *rules* applied to *states* in a sequence of steps assimilated to a *run*. States are structures in the sense of first-order logic, with relations treated as Boolean-valued functions. A finite collection of function names having a fixed arity is called a *vocabulary*, Γ. A state S of vocabulary Γ is a non-empty set X, together with the interpretation of all function names in Γ over X, therefore holding the values of all the variables at a specific step. *Updates* represent the simplest change that can occur to a state by the change of the interpretation of a function at one particular tuple of arguments. Let R be a rule that gives rise to a set of updates. In order to execute R at S all the updates are triggered in the corresponding update set. Thus we have the *update rule*, the *block rule*, a sequence of transitions rules that are executed simultaneously, the conditional rule **if** g **then** R_0 **else** R_1 **endif**, etc.

4.2 Hierarchical Timed ASM

Possessing a precise and versatile model of the processor is very important. Nevertheless having access to an usable HDL code, is rarely the case for platforms used in hard real-time systems, that are fairly outdated, and even if it exists, their is no common, unified description language. Ideally we should use the description of the processor as an input and generate an usable model for the analysis. As the lack of availability and standardization makes the task impossible, the need to create a model for each platform is mandatory. This is one of the bottlenecks in the adaptability of current tools. We consider that the modeling part should be therefore a separated, straightforward engineering task that can be made on the fly, without disposing of precise knowledge with regard to the rest of the tool. Therefore we chose to use the ASM, a model that bridges the gap between human understanding and formulation of real-world problems and the deployment of their algorithmic solutions, in our case, the modeling of the processor. The ASM showed its efficiency as a specification method in numerous practical applications (e.g. see [14], [15]).

Using a human readable and machine executable language makes the difference when it comes to speeding up the process of the hardware description. However some important features were not included in the original version of the ASMs [13] like the timing aspects hence updates are considered immediate. Ouimet et all. [16] introduced the concept of durative actions by adding delays directly in the syntax; our approach is similar. In [17] a prototype of a simulator for reactive timed ASMs that verifies the respect of requirements specifications. Besides the timing aspects we enrich the original model with hierarchical feature that enables us to give different definitions on several abstraction levels of the same processor component.

The goal of hierarchical ASMs is to provide at any time during the analysis, the right level of abstraction in order to prevent the combinatorial explosion. We know that we do not always dispose of precise information during the analysis (e.g. data memory address, availability in the cache, etc.). Therefore using the most precise component description, the fetching mechanism for example, would be useless, on the other hand, a less precise, more abstract, definition can help reduce the number of generated states.

The hierarchical definition of components integrates seamlessly into the ASM formalism. Basically, the *oracle* is an ASM module that imports all the necessary function definitions and exports the needed functions or rules. Each hierarchical module is defined as a control state ASM, cf. [15], using in its condition the result from the *oracle* that automatically decides which implementation is

appropriate for the current context. The *oracle* has several general strategies and it is further guided by information dependent on the current analyzed platform. A dynamic mode is also available that changes the default decision strategies based on the history of its success. However, having to compare different executions in parallel in order to confirm the strategies is costly and will only be used as a last resort. Equivalence classes for the data shuffle are under study in order to determine a pattern dependent on the degree of precision that we dispose on the data.

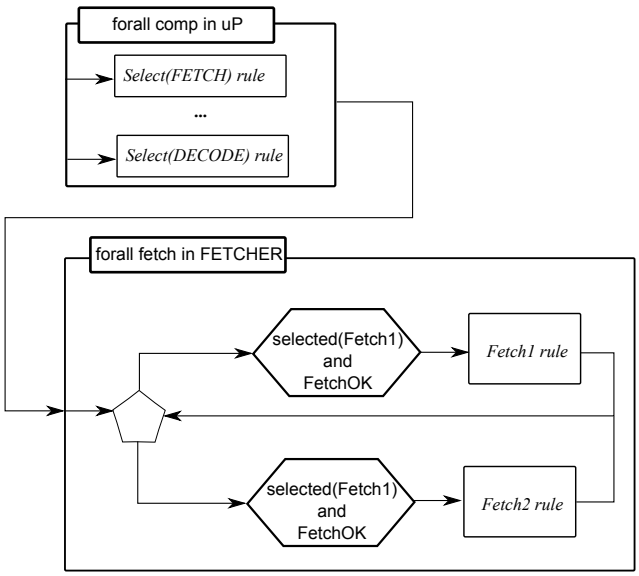


Figure 2. The *oracle* and the fetcher modules

Figure 2 shows the *oracle* and two fetcher models, pictorially depicted in a fashion inspired by the control state ASMs of Borger et all.,[15], equivalent to the ASM definition bellow.

```
FETCH =
forall fetch in FETCHER do
{FETCH1(fetch), FETCH2(fetch)}
```

In Figure 3 we have two definitions of the Fetch stage, the first one corresponding to the more abstract version that will typically be chosen by the *oracle* if we have no precise information on the exact fetch address. Generally we have a family of abstraction for each component of the processor, $\alpha_{C_i} = \bigcup_{j=0}^{m} \alpha_j$ so that $C_i \overset{\alpha_j}{\rightarrow} C_i^{\alpha_j}$. Let $T(C_i^{\alpha_j})$ be the contribution of the abstract component to the global execution time. We must have $T(C_i^{\alpha_j}) \sqsupseteq T(C_i)$.

5. Conjoint Symbolic Execution

The use of *SE* to analyze the intra-processor interactions has been implemented with good results in [11], however the method suffers from the lack of a precise hardware model and inaccurate merging strategies that lead to important overestimations. The basic *SE* consists in replacing the variables with symbolic values and extending the operations in order to take this into account. The interpretation of the assignment rule is straightforward. Let $p(pc)$ be Q, $p(x_i)$ be E_i and $p(\alpha \leftarrow \beta)$ be the old p where the value of α is changed to β. A special treatment is applied to conditional instructions that use the pc to explore all the possible scenarios. The expressions conjoined in the pc are of form $Q > 0$ where Q is a polynomial

```
FETCH
if FetchOK then
    FetchQueue:=getNextInstr()
    t:+=[t_min, t_max]
endif

if FetchOK then
   FetchAddr:=getExactFetchAddr()
   howMany:=FetchAddr MOD 4
   FetchQueue:=BurstAccess(
            FetchAddr,howMany)
   t:+=[t_BurstFetch]
endif
```

Figure 3. Different definitions of the fetcher

over symbolic values. Let R be this expression. We thus have three possible cases. We can determine from the pc that the condition is always true ($pc \supset R$ and $pc \not\supset \neg R$), analogue for always false or we can not determine if the condition is true or false, $pc \supset R$ and $pc \supset \neg R$, therefore the execution will continue along both branches, generating two new paths.

The first step of our conjoint *SE* deals with the program's CFG that is regarded as an input for the processor's model *SE*.

6. Smart State Fusion

One of the major drawbacks of the SE comes from its quality of generating every feasible path, that for a real-life industrial program generates a combinatorial explosion that is not obviously containable. What still remains challenging today is to handle this explosion while still remaining precise enough that translates to finding a good way of eliminating some of the states. We choose the technique of states fusion that will try to generate an abstract state capable of capturing the respective states features, with regards to the goal, but remain as compact as possible. It has been proven in [18] that because of the finite number of states a processor can have and because of the constrains generated by the execution contexts at a certain point we will have states that regardless of the different history, will generate identical or very similar new states. One major step in having precise fusions is to determine when to make them and what changes to apply. States can be of two types: identical, meaning that they have either all the elements that are the same, in this case we can suppose that an eventual fusion will not impact the precision of the analysis, or similar, some of the components are not the same so we proceed to another analysis to determine to which extent they are different. Therefore similar states can be strongly or weakly similar, meaning that the impact of the fusion will be acceptable or not. For the instant this estimation is done dynamically by our prediction module. Its goal is to evaluate the impact in the future of a fusion by unrolling the tree for several steps (generally equal to the pipeline depth), continuing the execution along the paths before and after fusion and comparing the result. Further details about this technique can be found in [18].

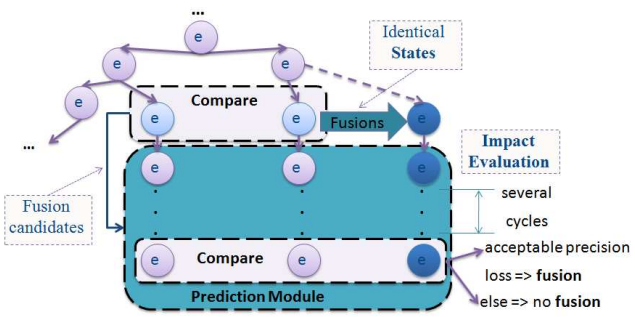


Figure 4. The Dynamic Fusion - snapshot of the Prediction Module

7. Global algorithm

1. Start from the initial state: where all the components have the unknown value and pc is set to $true$
2. For every variable that we encounter and that we do not have the exact value, assign a symbolic value
3. Activate the first ASM model and then add the guard condition g to the pc
4. Choose from the *oracle* the appropriate version of the ASM modules
5. Compute the update set of the current step
6. Apply the update set (taking into account that some terms will have symbolic values)
7. Add the result of the update set to the global system state
8. Add the generated states to the collection of next states to be executed
9. Add the duration of the transition to the global time
10. Repeat from point 2. until the collection of next states is empty

8. Conclusions

The world of embedded software is no longer integrating simple hardware/software therefore critical systems are becoming more and more difficult to prove and certify. The growth in complexity and variety increases the need of versatile analyze methods and adapted tools, that can easily and as costless as possible deal with a large panel of architectures. To this end we presented a novel approach that is able to respond to the evergrowing demands and to place itself into a real industrial context.

References

[1] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, P. Stenstrom The Worst-Case Execution Time Problem Overview of Methods and Survey of Tools, *ACM Transactions on Embedded Computing Systems (TECS)*, Volume 7, Issue 3, April 2008.

[2] T. Lundqvist and P. Stenstrom, An Integrated Path and Timing Analysis Method based on Cycle-Level Symbolic Execution, in *Real-Time Systems*, Volume 17, 183-207, November 1999.

[3] J. Teich, P. W. Kutter, and R. Weper, Description and Simulation of Microprocessor Instruction Sets Using ASMs, in Proceedings of the International Workshop on Abstract State Machines, Theory and Applications (*ASM '00*), (Eds.). Springer-Verlag, London, UK, 266-286, 2000.

[4] G. Schellhorn, Verification of ASM Refinements Using Generalized Forward Simulation, *Journal of Universal Computer Science* (J.UCS), 7(11):952-979, 2001.

[5] K. Winter, Model Checking Abstract State Machine, *PhD Thesis*, 2001.

[6] G. Castillo, Towards comprehensive tool support for Abstract State Machines: The ASM Workbench tool environment and architecture, in D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, eds., Applied Formal Methods - *FM-Trends 98*, Springer LNCS 1641, 311–325, 1999.

[7] H. H. Daho and D. Benhamamouch, Formal Verification of ASM Models Using TLA +, in Proceedings of the 1st international conference on Abstract State Machines, B and Z (*ABZ '08*), (Eds.). Springer-Verlag, 2008.

[8] H. Casse and P. Sainrat, Otawa, A framework for experimenting WCET computations, *ERTS06*, 2006.

[9] R. Wilhelm, Formal Analysis of Processor Timing Models, in Proceedings of the 11th SPIN Workshop Barcelona, Spain, 2004.

[10] R. Heckmann, C. Ferdinand, Worst-Case Execution Time Prediction by Static Program Analysis, http://www.absint.com.

[11] T. Lundqvist, A WCET Analysis Method for Pipelined Microprocessors with Cache Memories, Goteborg, Sweeden, 2002.

[12] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker, A vDefinition and Classification of Timing Anomalies, *WCET06*, 2006.

[13] Yuri Gurevich, Evolving Algebras 1993: Lipari Guide, Specification and Validation Methods, ed. E. Brger, *Oxford University Press*, 9–36, 1995.

[14] University of Michigan, ASM homepage. http://www.eecs.umich.edu/gasm/.

[15] E. Borger and R. Stark, Abstract State Machines: A Method for High-Level System Design and Analysis, *Springer-Verlag*, 2003.

[16] M. Ouimet and K. Lundqvist, The Timed Abstract State Machine Language: Abstract State Machines for Real-Time System Engineering, *JUCS*, 2007.

[17] A. Slissenko and P. Vasilyev, Simulation of Timed Abstract State Machines with Predicate Logic Model-Checking, *JUCS*, 2008.

[18] Bilel Benhamamouch, Bruno Monsuez: Computing worst case execution time (wcet) by symbolically executing a time-accurate hardware model (extented version), *International Journal of Design, Analysis and Tools for Circuits and Systems*, Volume 1, No. 1, November 2009.

WCET Estimation of Multi-Core Processors with the MSI Cache Coherency Protocol

Pradeep Subedi and Wei Zhang
Electrical and Computer Engineering
Virginia Commonwealth University
Richmond, VA 23284
Wzhang4@vcu.edu

Abstract

To safely exploit multi-core processors for hard real-time systems, it is a necessity to be able to estimate the worst case execution time (WCET) of parallel programs running on a multi-core processor. For parallel programs sharing data, the cache-coherency protocol used in a multicore processor may turn an otherwise cache hit into "invalidated" or a miss, making it hard to safely estimate the WCET. In this paper, we focus on studying a multi-core processor with the cache coherency protocol MSI (Modified, Shared and Invalid). We present an approach to extend the abstract interpretation technique to model and statically analyze additional states caused by the MSI protocol. Our approach can safely estimate the WCET of parallel programs running on a multicore processor with the MSI protocol.

Categories and Subject Descriptors

C.3 [**Special-purpose and Application-based Systems**]: Real-time and Embedded Systems

General Terms

Design, Performance.

Keywords
WCET, Multi-core

1. Introduction

In hard real-time systems, it is crucial to compute the upper bound of the execution time of a real-time task [1], also known as worst-case execution time (WCET). One method to estimate the WCET is to exhaust all the possible program paths, for a given input through measurement, but this method may be infeasible for large programs or complex architectures with a large set of initial states. Another method is to obtain the WCET by static analysis of the program. The static analysis of the sequential programs running on a uniprocessor is done by finding the longest feasible path in the program's control flow, and takes into consideration the timing of the micro-architectural components of the system [3, 5]. For the parallel programs running on a multi-core processor with a cache-coherency protocol, in addition to considering the resource contention and inter-thread conflicts among the program threads, it becomes crucial to calculate the worst-case delay caused by maintaining the cache coherence between different cores.

Many research efforts on WCET analysis of multi-core processors [6, 7] have focused on bounding the inter-core cache interferences in the shared caches. However, to the best of our knowledge, no prior work has studied the effect of cache-coherency in WCET estimation, which may be unsafe for real-time applications that have concurrent threads sharing data. By comparison, this paper examines a timing analysis method to safely estimate the WCET of parallel programs running on a multi-core processor with the MSI cache coherency protocol.

2. System and Application Model

In a multi-core processor, each core usually has a private L1 instruction cache and an L1 data cache. The L2 cache can be either shared or private. In this work, we assume a dual-core processor with a shared L2 cache as shown in Figure 1. We assume a real-time thread and another real-time or non-real-time thread are running concurrently on these two cores, potentially sharing some data through both read (i.e. load) and write (i.e. store) operations. The MSI protocol [9] is used for ensuring cache coherency.

The MSI protocol has three states: Modified, Shared and Invalid. If a cache block is in the modified state, the cache block is valid in only one cache, and it implies exclusive ownership of the cache. A shared cache block implies that the cache block is valid, and maybe shared by multiple processors. The Invalid state of a cache block means that the copy of the data in the current cache block is outdated (because another thread from another core has modified it) and it must be updated from the shared cache or the memory. More details about the MSI can be found in [9].

Due to the existence of different cache states, the worst-case delay of memory accesses and eventually the WCET can be affected. For instance, even if the cache analysis indicates an access

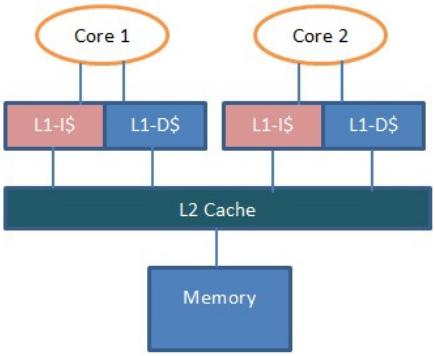


Figure 1. A dual-core processor with shared L2 cache

LCTES'12 June 12–13, 2012, Beijing, China.

may be always a "hit", if the state of this cache block is "Invalid" due to a previous write by another concurrent thread, this otherwise "hit" will basically be converted into a miss by reloading the data from the lower-level memory for keeping data coherent. Therefore, we need a new method that is aware of the cache coherency protocols to safely estimate the WCET for multicores.

3. Analysis Framework

In this section, we will present an overview of the analysis framework for a multi-core processor with the MSI cache coherency protocol. Figure 2 overviews our analysis framework, which involves three major steps, including the address evaluation, the cache analysis with the MSI protocol, and the WCET estimation.

3.1 Assumptions

First, we assume that the loop bounds are known through either static analysis or user annotation. Second, we assume the program has no infeasible paths. Third, we assume that the multicore processor uses the Least Recently Used (LRU) cache replacement policy and the MSI protocol for cache coherency. Forth, we assume there is no timing anomaly or domino effect.

To focus on cache analysis, we do not model the shared bus in this work. In our analysis, we use a shared address array $\hat{X}$ to model all the memory blocks that are shared by different cores. We associate a tag to each shared memory block, which indicates whether a core has recently modified any data or not. More specifically, if the tag of a memory block m in $\hat{X}$ is *I1* (or *I2*), it means it is modified by core 2 (or core 1) and thus the corresponding data in core 1 (or core 2) become invalid. And if the state is *s*, it indicates the block is in the shared state.

3.2 Address Evaluation

Since we analyze both the data and the instruction caches, we need to get the set of addresses accessed by the given task. Instruction references are relatively straightforward to analyze [6]. For data references, we use the register expansion framework [2] to identify the relative addresses of all memory blocks. For each register that is used to specify the address of a *load* or *store* instruction, we perform register expansion recursively to trace the source registers and computation is performed to evaluate the set of memory blocks accessed by that instruction. If a *load* instruction and a *store* instruction are mapped to the same memory block, they can potentially cause cache coherency misses unless one can statically prove that the *load* occurs before the *store*.

Based on the assembly codes of two tasks (or threads) task 1 and task 2 running on the core 1 and core 2 respectively, two control flow graphs are created and analyzed. First we analyze all the *load* and *store* instructions in tasks 1 and 2. Then the set of addresses accessed by task 1 and task 2 are stored in two arrays. If any two sets of addresses are overlapped, then our algorithm identifies that they point to a shared address in the main memory, which is then added into the shared address array $\hat{X}$.

3.3 Cache Coherency Aware Analysis of Caches

For cache analysis, the abstract interpretation method is used. Although we use the method presented in [5], we make some changes to the method to include the effect of the cache coherency. A memory block m in the shared array $\hat{X}$ is represented as m_j, where j represents the status of the block and can be *I2* (i.e., modified by the task running on core 1, thus invalid for the task running on core 2), *I1* (i.e., modified by the task running on core 2, thus invalid for the task running on core 1), or *S* (i.e., shared). The

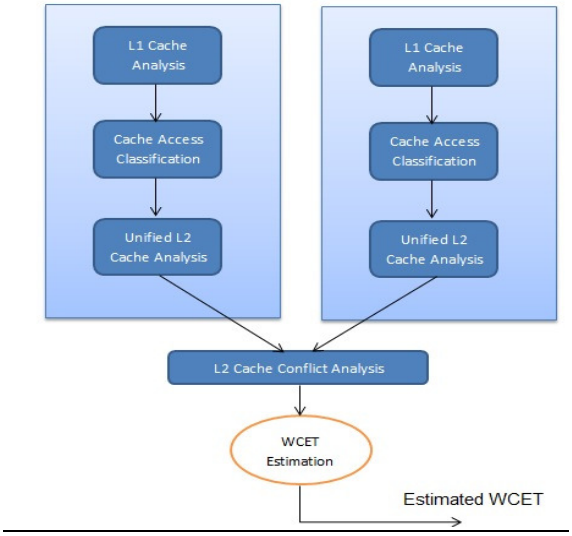


Figure 2. Our analysis framework

function $shared(\hat{X}, m)$ takes a memory block m and the shared array $\hat{X}$ as inputs and returns the state of the memory block in the shared array. For instance, if a memory block m_0 is in the shared array and its state is *S*, then the above mentioned function returns *S*, meaning the data is in the shared state in the cache.

Algorithm 1 shows how the states inside the shared array $\hat{X}$ are changed. This algorithm is run after the evaluation of the shared variables. The initial state of all the shared memory blocks is assumed to be x (i.e. unknown).

Algorithm 1. Change of states inside shared array $\hat{X}$, with the read and write operations

```
if (analyzing in core 1) then
    for (all instructions of the task running on core 1)
      if (the instruction is read) then
        st = state of a memory block possibly referenced by the instruction
        if (st is x or I1) then
              set states of all memory blocks possibly referenced by this
              instruction to S;
        end if
      end if
          if (instruction is write) then
              set states of all memory blocks possibly referenced by this
              instruction to I2;
          end if
    end for
end if
if (analyzing in core 2) then
    for (all instructions of the task running on core 2)
      if (the instruction is read) then
        st = state of a memory block possibly referenced by the instruction
          if (st is x or I2) then
              set states of all memory blocks possibly referenced by this
              instruction to S;
          end if
        end if
        if (the instruction is write) then
              set states of all memory blocks possibly referenced by this
              instruction to I1;
        end if
    end for
end if
```

For a task running on core 2, the state in $\hat{X}$ is also changed as mentioned in the algorithm. For every write the task sets the state to *I1* and for read operations accessing memory blocks having state *I2* (in $\hat{X}$), the state is changed to *S* and the data is fetched from the lower-level memory.

In the following, we consider an A-way set associative cache with a number of cache sets F=< $f_1,\ldots,f_{n/A}$ > where n = capacity/line size, a cache set f_i consisting of A cache lines f_i = < $l_1,\ldots,l_A$>, and all the store operations to this set represented by a set of memory blocks M= {$m_1,\ldots,m_s$}.

The function $set: M \mapsto F$ determines the cache set where a memory block will be stored (% denotes the modulo addition): $set(m) = f_i$; where i= adr(m) %(n/A) +1. The function adr: M $\mapsto N_0$ gives the address of each memory block [4].

The cache update function is a state update function that models the LRU replacement strategy. The following function takes a cache set and a block and returns the updated cache as follows.

$$u_s(s, m_j) = \begin{cases} [l_1 \mapsto m \\ l_i \mapsto s(l_{i-1}) | \, i = 2 \ldots h \\ l_i \mapsto s(l_i) | \, i = h+1 \ldots A]; \\ \qquad if \; \exists l_h : s(l_h) = m \\ \\ [l_1 \mapsto m \\ l_i \mapsto s(l_{i-1}) | \, i = 2 \ldots A\}; \\ \qquad otherwise \end{cases}$$

The notation [y ↦ z] denotes a function that maps y to z.

The abstract set state $\hat{s}: L' \mapsto 2^{M'}$ maps set lines to a set of memory blocks. An abstract cache state $\hat{c}: F \mapsto \hat{S}$ maps sets to abstract set states. $\hat{S}$ denotes the set of all abstract set states and $\hat{C}$ denotes the set of all abstract cache states.

The abstract update function is modeled as

$$\hat{u}_{\hat{C}}(\hat{c}, m) = \hat{c}[set(m) \mapsto \hat{u}_{\hat{s}}\left(\left(\hat{c}(set(m)), m\right)\right)]$$

The join functions that we use for must, may, and persistence analysis are based on [4].The join function for must analysis is similar to set intersection, except that if a memory blocks has two different ages in two abstract cache states, the join function takes the oldest age.

$$\hat{J}_{\hat{S}}(\hat{s}_1, \hat{s}_2) = \hat{s}$$

where,

$$\hat{s}(l_x) = \{m | \exists l_a, l_b with \; m \in \hat{s}_1(l_a), m \in \hat{s}_2(l_b) \; and \; x = \max(a, b)\} \\ \cap \{m | \, m \in \hat{s}_1(l_x) and \nexists l_a with \; m \in \hat{s}_2(l_a)\} \\ \cap \{m | \, m \in \hat{s}_1(l_x) and \nexists l_a with \; m \in \hat{s}_2(l_a)\}$$

The join function for may analysis and persistence analysis are similar to the above except that in may analysis, set union is performed and the minimum age is taken, and in persistence analysis set union is used and the maximum age is taken. The abstract interpretation method [5] is used at both L1 and L2.

For the level 1 instruction and data caches, *must*, *may* and *persistence analyses* are firstly conducted. In case of an instruction cache, the analysis is done as specified in [5]. The L2 cache analysis is similar to the L1 cache analysis except that it does not take into consideration the blocks that are classified as AH (i.e. Always Hits [5]) in L1 caches.

The references are provided with Always (A), Never (N), and Uncertain (U) tags for the L2 cache analysis.

L1 Classification	L2 tag
AH	N
AM	A
NC	U

Table 1. L2 tag for different L1 classification of memory blocks

If a reference tag is N, the abstract cache state is not updated because they do not access the L2 cache at all. For the reference with tag U, two abstract cache states are created; one updates the reference while another does not update the reference and both of them are then joined later.

Since the L2 cache is a unified cache, for each instruction, initially the abstract cache state is updated by using the update function of the instruction cache analysis first. If the instruction is a *load/store* instruction, and it references a set of memory blocks M, then the result of the update function for instruction cache is updated by the update function of the data cache analysis.

Now we have to take into consideration the cache conflicts in L2 because it is shared and a task T2 executing on core 2 can potentially conflict with task T1 executing on core 1. This brings changes in the hit/miss classification of the memory blocks in T1. Since the references with N tag never access L2 cache, we need not consider these references for the conflict analysis. Now if the memory block in T1 is classified as AM or NC, then the cache state remains the same because the interfering task cannot downgrade it further. So only the memory blocks with AH are affected. Since the memory blocks can be evicted from the cache, we classify these interfering memory blocks as Non-classified (NC). For set-associative caches, the method introduced in [7] can be used for conflict analysis, where the age of the memory block is taken into consideration and if the number of conflicting memory blocks from the conflicting tasks is less than or equal to the *N-age* of memory block in T1, where *N* is the associativity of the cache. The cache state of the memory block is not modified because it will not be evicted from the cache, resulting in the AH for the L2 cache. The age of a memory block is equal to the line number of the cache block that the memory block is mapped into.

3.4 WCET estimation

Table 2 shows the worst-case access latency for a reference that is taken into consideration during the WCET estimation for a basic block, where hit_{L1} is the latency of a hit at the level 1 cache; hit_{L2} is the latency due to a miss in the L1 cache but a hit in the level 2 cache; and $miss_{L2}$ is the latency due to misses in both the level 1 and the level 2 caches (i.e., the data must be fetched from the main memory). After finding out the worst case latencies for all the memory references, our algorithm then sums up these worst case latencies to derive the WCET of caches without considering the cache coherency misses.

L1 cache	L2 cache	Worst-case Access Latency
AH	-	hit_{L1}
AM	AH	hit_{L2}
AM	AM	$miss_{L2}$
AM	NC	$miss_{L2}$
NC	AH	hit_{L2}
NC	AM	$miss_{L2}$
NC	NC	$miss_{L2}$

Table 2. Access Latency of a reference in the worst case given its classification

To safely estimate WCET for real-time tasks that share data across different cores, we must incorporate the delay caused by cache coherency misses. Algorithm 2 described below presents our method of finding the worst-case delay caused by all the invalidations inside a basic block, which provides an important basis to derive the WCET for a multicore processor with the MSI protocol.

Algorithm 2. Latency caused by invalidation of memory block m inside a basic block b; shared $(\hat{X}, m)$ returns the state of memory block in shared array; *I1* means invalid for core 1; and *I2* means invalid for core 2.

```
cost_invalidation =0;
  invalidation=0;
  inst= first instruction of the basic block;
  repeat
    if (inst is a read instruction) then
          n = number of memory blocks possibly accessed by inst;
          if (n==1) then
              if(shared (X̂, m)==I1) then
                if (analyzing core 1 and reference classification is
                not (AM for L2 or NC for both L1 and L2) ) then
                  invalidation=invalidation+1;
                end if
              else
                if (shared (X̂, m)==I2) then
                        if (analyzing core 2 and reference classi-
                        fication is not (AM for L2 or NC for
                        both L1 and L2)) then
                        invalidation=invalidation+1;
                       end if
                 end if
          end if
        else
          if (n>1) then
            m_k = any one memory block possibly referenced;
            for(all possible m_k)
            if(shared (X̂, m_k)==I1) then
                if (analyzing core 1 and reference classification is
                not (AM for L2 or NC for both L1 and L2)) then
                  invalidation= invalidation+1;
                end if
            else
                 if (shared (X̂, m_k)==I2) then
                    if (analyzing core 2 and reference classifica-
                    tion is not (AM for L2 or NC for both L1 and
                    L2)) then
                        invalidation= invalidation+1;
                    end if
                end if
             end if
           end if
         end if
    end if
  until (all instructions in basic block b finish)
  cost_invalidation = invalidation*miss_L2 latency;
  return cost_invalidation
```

Suppose lat_b is the worst case latency for a basic block b without considering invalidation and b_i is the number of execution times of basic block i, then total cost for that basic block can be computed as the follows, where $cost_{invalidation}$ can be calculated by using the Algorithm 2.

$$\sum (lat_b + cost_{invalidation}) \times b_i$$

Now the longest path search [8] or the implicit path enumeration technique (IPET) [10] can then be applied to obtain the WCET of the whole thread.

4. Concluding Remarks and Future Work

In this paper, we have presented the challenge of timing analysis for multicore processors with cache coherency protocols. To address this problem, we extend the abstract interpretation technique to model additional cache states, which can safely derive the worst-case cache performance by considering coherent cache misses. Our ongoing work includes implementation and validation of the proposed static analysis, and the exploration of additional techniques to improve the tightness of analysis, for example, using timing overlapping information among concurrent threads to reduce overestimation.

References

[1] R. Wilhelm *et al., The worst-case execution-time problem-overview of methods and survey of tools.* Trans. Emb. Comp. Syst., vol. 7, no. 3, 2008.

[2] G. Balakrishnan and T.W. Reps., *Analyzing memory accesses in x86 executables.* CC, 2004.

[3] S. Chattopadhyay, A. Roychoudhury., *Unified Cache Modeling for WCET Analysis and Layout Optimizations.* Proceedings of the 2009 30th IEEE Real-Time Systems Symposium, p.47-56, December 01-04, 2009.

[4] M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm., *Cache behaviour prediction by abstract interpretation.* SAS. Springer-Verlag, 1996.

[5] H. Theiling, C. Ferdinand, and R. Wilhelm. *Fast and precise WCET prediction by separated cache and path analyses.* Real-Time- Systems, 18(2/3), 2000.

[6] S. Chattopadhyay, A. Roychoudhury and T. Mitra., *Modeling Shared Cache and Bus in Multi-cores for Timing Analysis.* 13th International Workshop on Software and Compilers for Embedded Systems (SCOPES), 2010.

[7] Y. Li, Vivy S., Yun Liang, T. Mitra and A.Roychoudhury., *Timing Analysis of Concurrent Programs Running on Shared Cache Multi-cores.* IEEE Real-time System Symposium (RTSS), 2009.

[8] V. Suhendra, T. Mitra, A. Roychaudhary, and T. Chen. *Efficient detection and exploitation of infeasible paths for software timing analysis.* Proceedings of the Design Automation Conference, 2006.

[9] Yan Solihin, *Fundamentals of Parallel Computer Architecture,* Solihin Publishing and Consulting LLC, 2009.

[10] Y. S. Li and S. Malik. *Performance analysis of embedded software using implicit path enumeration*. In Proc. of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Real-Time Systems, 1995.

Acknowledgement

This work is partially supported by NSF grant CCF 0914543.

Introducing Service-oriented Concepts into Reconfigurable MPSoC on FPGA for Coarse-grained Parallelization

Chao Wang[1,2+] Xi Li[1] Peng Chen[2] Junneng Zhang[2] and Xuehai Zhou[1]
[1]University of Science and Technology of China, Hefei, Anhui China, 230027
[2]Suzhou Institute of Advance Study, USTC, Suzhou, Jiangsu, China, 215123
{saintwc,zjneng, qwe123}@mail.ustc.edu.cn {llxx,xhzhou}@ustc.edu.cn

ABSTRACT

High level programming into hardware is posing significant challenge for reconfigurable modular embedded systems. In this paper we propose SOREP: a Service-oriented Reconfigurable Prototype, which introduces service-oriented architecture (SOA) concepts to reconfigurable multiprocessor system on chip (MPSoC) on field programming gate arrays (FPGA) for coarse-grained parallelization. SOA concepts can provide a uniform programming model as well as computing resources integration manners to MPSoC hardware. With the benefits of SOA and state-of-the-art reconfigurable technologies, novel MPSoC design paradigms are encountering new opportunities for traditional technical challenges, including reconfigurable task execution, programming models and out-of-order scheduling. For demonstration, a service-oriented reconfigurable MPSoC prototype has been built on FPGA, regarding embedded processors and IP cores as computing servants. The preliminary results demonstrate the prototype can achieve more than 95% of the theoretical peak speedup on average.

Categories and Subject Descriptors

D.4.1 [**Process Management**]: Multiprocessing/ multiprogramming / multitasking

General Terms Algorithms, Design

Keywords
Service-oriented architecture, multiprocessor system-on-chip, reconfigurable computing,

1. Introduction and Motivation

FPGA based reconfigurable MPSoC has been considered as one of the promising future microprocessor design paradigms [1]. However, current MPSoC developers are still suffering from limited programming ability and high complexity during the design of different architectures for various applications [2]. Whenever hardware is reconfigured, developers need to redesign the middleware, programming models or even the tool chains. Moreover, runtime task partition and scheduling schemes need also to be carefully reconsidered, which in case could dramatically drag down the degree of task parallelization. In order to address the above challenges, our research tentatively introduces service-oriented concepts into reconfigurable MPSoC.

SOA concepts have been successfully applied in software engineering and web services, and it's shifting towards lower level, such as operating systems [3]. However, to our best knowledge, so far, few existing literature have been conducted to introduce SOA to reconfigurable hardware architecture designs. Instead, most state-of-the-art FPGA based research platforms are constructed in specific hardware, which means users need to acquire the specific hardware configurations and scheduling schemes of the system to handle the tasks distribution manually. Otherwise, the parallelization degree could be dramatically dragged down due to unsophisticated programmer's experiences. Therefore, the automatic parallelization degrees are still worth pursuing.

From the weakness exploration of current studies, **the motivation of this paper is to integrate SOA with MPSoC to absorb the advantages of both concepts.** From the exploration of SOA concepts' benefits, we can conclude that there are at least two significant advantages through integrating SOA to MPSoC platform. Firstly, servant integration interfaces are well defined, which facilitate researchers to add/remove modularized function units expeditiously during prototype system construction. Secondly, with unified API, users are no longer concerned about the target hardware, which means the partition and mapping is handled by SOREP automatically. This feature will significantly ease the burden of programmers, shorten MPSoC design cycle, and reduce the complexity to construct a heterogeneous single chip cloud.

In this paper, we tentatively propose a prototype to demonstrate SOA concepts into heterogeneous reconfigurable multi-core platform design. Based on the previous research of [4], this paper builds a hardware prototype on FPGA with state-of-the-art dynamic partial reconfigurable technologies. **We claim following contributions:**

1) This paper brings SOA concepts into real reconfigurable MPSoC hardware, and builds a service-oriented reconfigurable prototype SOREP. SOREP is implemented on state-of-the-art Xilinx FPGA with multiple Microblaze processors and adaptable IP cores. The reconfigurable characteristics demonstrate the high flexibility and scalability of SOREP.

2) SOREP provides an efficient experimental research platform to attack traditional key challenges including IP reconfiguration, task partition, and out-of-order task scheduling. First, a self reconfigurable task execution model based on state-of-the-art Xilinx Early Access Partial Reconfiguration (EAPR) is carried out to minimize the reconfiguration overheads. Second, when the reconfiguration is ready, tasks are adaptively remapped to computing servants without rewriting or recompilation. Third, we also apply renaming techniques from instruction level to chip level which can automatically detect inter-task data hazards, and then distribute tasks to computing servants for out-of-order execution.

The novelty of SOREP against current state-of-the-art MPSoC architectures are listed as follows:

1) SOA concepts bring unified programming models and servant integration interfaces, which can largely facilitate researchers to construct flexible experimental platforms. Therefore developers can concentrate on the key issues of scheduling, on chip interconnection methodologies, and reconfigurable technologies, etc. SOREP can ease the burden of MPSoC architects and shorten the time to market (TTM) of chips.

2) SOREP maintains middleware to fully support automatic parallelization, including adaptive task partition and out-of-order scheduling. In our approach, the middleware is designed under the IP dynamic reconfigurable condition. We integrate those methods into a SOREP framework, which can be easily customized to build an application-specific MPSoC.

2. SOREP Architecture and Concepts

Before SORA architecture is introduced, we define the following terms at first.

Service: A service is defined as a specific kind of functions with programming interfaces. All services are packaged into libraries and can be invoked by function calls.

Servants: Servants refer to functional modules dedicated to provide services to run specific tasks in hardware. All servants are IP cores packaged in same manners.

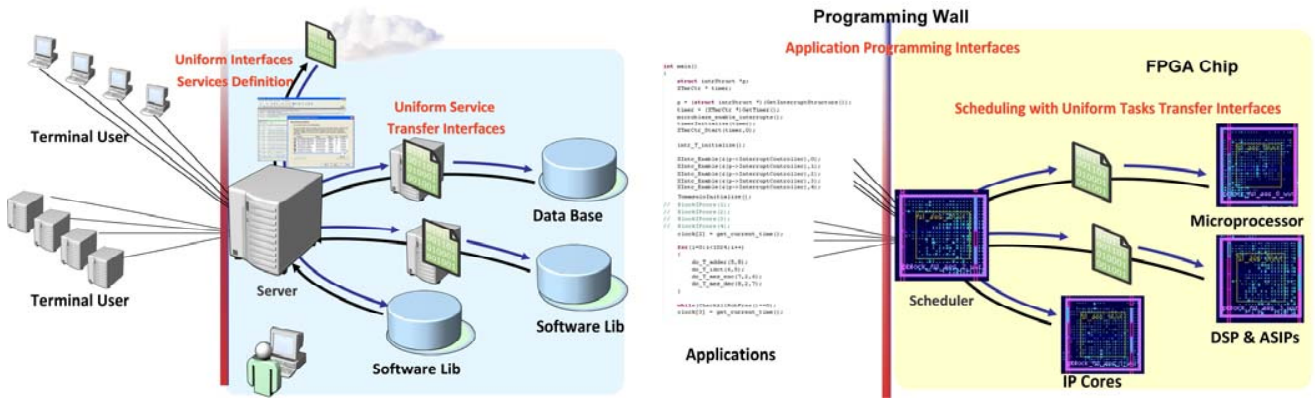
Figure 1 [a] illustrates the similarity of traditional SOA concepts. All services are provided via front-end uniform interfaces with service definitions. In the back stage, each service is composed of specific functionalities transferred from software libraries and data bases through the uniform service interfaces. Similarly, the situation on MPSoC is presented in Figure 1 [b], where the service definition interfaces are turned into the application programming interfaces (APIs), and each service providers are turned into either a microprocessor, or a DSP/hardware IP core.

SOREP architecture consists of servants classified in two categories: one Scheduling Servants is employed for task partitioning, mapping and run-time scheduling. As the kernel component, scheduling servant also plays a key role in exploration for inter-task data dependencies. It can be regarded as non-scalable as we move from one to ten to hundreds of cores. However, it is possible to instantiate additional scheduling servants if the architecture scale is increased to more than 8 cores. Computing Servants are designed to provide computing services and can be further classified into hardware and software servants. Software servants run on microprocessors with libraries, while hardware servants are implemented in IP cores to run only one specific kind of tasks.

The scheduling servant is connected to computing servants and peripheral modules through via FSL channels. All the hardware servants are loaded from IP libraries dynamically. The middleware is composed of three levels: application, middleware layer and communication layer.

2.1 Application layer

Application layer consists of service API, run-time libraries, and application profiling. In order to maintain the consistent user programming behaviors, system should provide unified high-level abstract interfaces. The entire API is utilized for spawning computational tasks and receiving results from scheduling servant to computing servants. Moreover, in order to fully benefit from the advantages of the self-reconfiguration techniques, APIs are be kept unchanged after hardware reconfiguration. In order to prevent the considerable overheads of maintaining memory consistency,



[a] Typical Service-Oriented Architecture [b] Mapping Services to MPSoC

Figure 1. Typical Service-Oriented Architecture and Services model on MPSoC

we utilize message passing mechanisms like MPI). Two types of primitives are provided: blocking interfaces will stall the execution and wait until results return, while non-blocking interfaces will continue, results will return through interrupts.

2.2 Middleware layer

In this Section, we give an overview of the services middleware layer, which includes task partitioning, scheduling and reconfigurable task execution models.

(1) Automatic task partitioning

The automatic task partition methods supervise the procedure of how a single task is divided and then mapped to IP core. As each IP core can run only one kind of task, a task-to-servant table is employed to identify the target servant for each task. The table maintains a mapping of tasks to servants to virtualize the selection of the destination core. Each table entry contains the task ID currently running on that core as well as a count of the number of issued tasks destined for that core.

(2) Out-of-order task scheduling

Scheduling servant is in charge of when the task can be issued. For tasks which are independent from each other, they can be issued simultaneously. However, in many cases, inter-task data hazards (such as RAW hazards) make the tasks run in sequence.

In order to fully exploit the potential task level parallelism, we demonstrate traditional Scoreboarding and Tomasulo algorithms from instruction level to chip level for out-of-order task execution. Both techniques are common knowledge in textbooks used to detect data hazards automatically, while Tomasulo also can further eliminate WAW hazards by renaming technologies. Since the two technologies are quite familiar to architectural researchers, we will directly show some experimental results in Section 4.

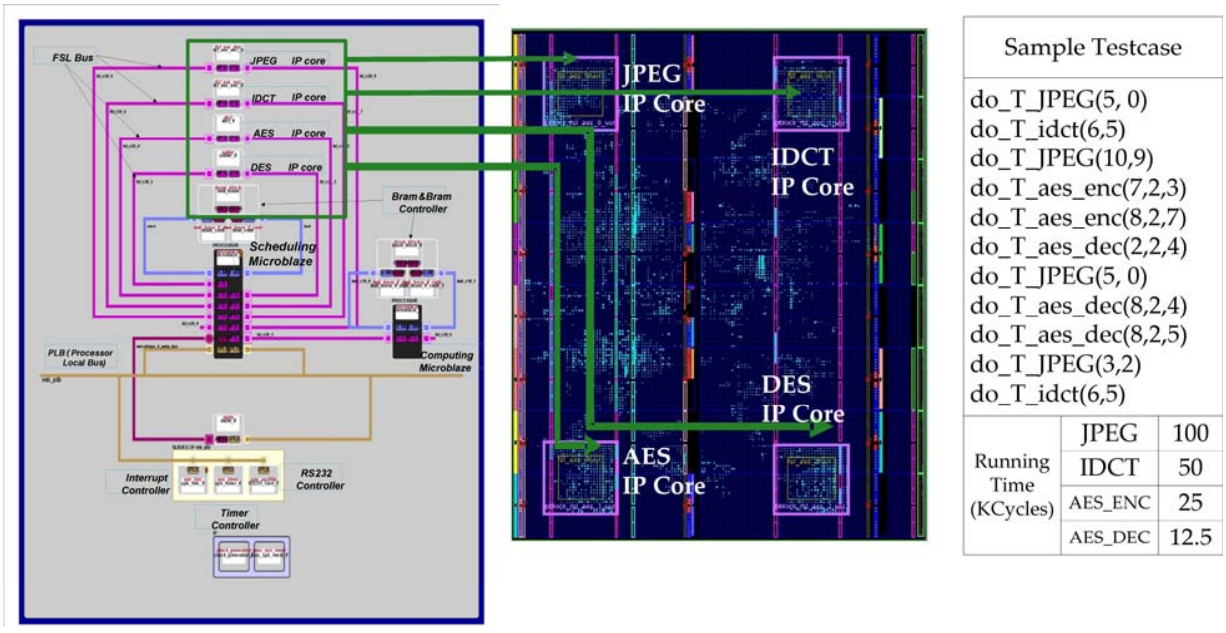
(3) Reconfigurable task execution model

At start-up, certain IP cores are loaded as servants in prior to task execution to provide an initialized run-time environment. When the target servant is ready, it can receive service requests from scheduling servant, run the task, and then return the result by raising an interrupt. However, if the target is not loaded during execution, current hardware should be reconfigured dynamically. Furthermore, if there are no more free areas in the chip, some of the present IP cores should be switched out. FIFO or LRU policies can be utilized under different conditions. For demonstration, self reconfiguration technologies based on Xilinx EAPR are introduced to support servant replacement.

3. SOREP Prototype on FPGA

We implemented a prototype for SOREP on a state-of-art Xilinx Virtex-5 XC5VLX110T FPGA. One Microblaze processor is used for scheduling servant, and another Microblaze processor is employed as software computing servant. Also, four hardware computing servants were integrated, including Data Accumulation (adder), and three EEMBC DENBench test cases of IDCT, AES_ENC and AES_DEC. For each service, one software computing servant is designed in C library running on Microblaze, and one hardware computing servant is described in HDL and packaged as IP core.

Figure 2 presents the self reconfigurable prototype in single FPGA. We integrate four hardware servants which can be substituted to other servants in the IP library. The reconfiguration procedure is manipulated by an internal controller without user interaction. Based on the hardware prototype, Figure 2 also gives a sample test case with the running time for each type of single task. The running time for each type of task is given in Figure 2. When the specific API is invoked during execution, an automatic adaptive



Sample Testcase
do_T_JPEG(5, 0)
do_T_idct(6,5)
do_T_JPEG(10,9)
do_T_aes_enc(7,2,3)
do_T_aes_enc(8,2,7)
do_T_aes_dec(2,2,4)
do_T_JPEG(5, 0)
do_T_aes_dec(8,2,4)
do_T_aes_dec(8,2,5)
do_T_JPEG(3,2)
do_T_idct(6,5)

Running Time (KCycles)		
	JPEG	100
	IDCT	50
	AES_ENC	25
	AES_DEC	12.5

Figure 2. SOREP prototype built in single FPGA and sample applications

mapping scheme will decide the target computing servant for each task. In this paper, due to the page limitations, we only give a demonstrative experiment to compare to mechanisms in parts of the total design.

All applications are running on the scheduling servant at first and then scheduled to certain computing servant. For demonstration, we use a greedy strategy: when there is available hardware computing servants, the task will be sent to hardware. Unfortunately, if all the hardware servants are occupied, the task will be dispatched to software computing servants. If all the software is also occupied, then SOREP can only wait for other applications to complete and first tries to schedule the application in hardware. This method is considered to show the effectiveness of partition scheme, and can be replaced by other algorithms to get a better result.

Within the hardware and programming interfaces, we measured the performance and accuracy of the FPGA platform, with the Scoreboarding and Tomasulo scheduling algorithms respectively.

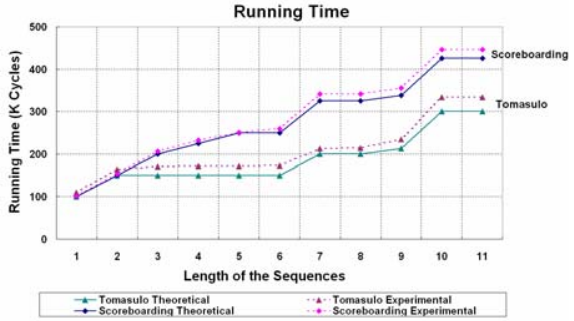


Figure 3. Comparison between Scoreboarding and Tomasulo

The empirical result of the test case in Figure 2 is presented in Figure 3. The length of sequences N indicates the first N tasks listed in Figure 2. The empirical results are quite close to the theoretical speedup, with most of the results more than 95% accuracy. In our scope, this result shows that Scoreboarding and Tomasulo algorithms can run tasks out-of-order with lower overheads than Task superscalar [5]. Compared to Scoreboarding, Tomasulo has higher scheduling overheads, which leads to a bigger gap between experimental and theoretical value. However, since Tomasulo can not only detect WAW and WAR hazards but also eliminate them by register renaming, the overall speedup is significantly larger than Scoreboarding.

We have also demonstrated the self-reconfigurable MPSoC prototype with Xilinx EAPR methods, and more than one IP cores can be reconfigured at run-time, with the FPGA chip-programming time at 409 ms approximately (For AES and DES IP modules).

In the prototype, we have integrated two Microblaze processors and four servants, as well as peripheral blocks. The system has been synthesized and Table I outlines the hardware cost for the entire system in single FPGA. SORA system takes 26% area in LUTs and 7% in registers overall, which means we can integrate more than 20 servants at the same time. Considering the abundant hardware resources supplied in FPGA, the resources are acceptable to construct a heterogeneous CMP.

TABLE I. SYSTEM HARDWARE COSTS OF THE ARCHITECTURE

Resource	Status	Percent
Number of Slice Registers	6493 out of 17280	7.00%
Number of Slice LUTS	18209 out of 69120	26%
Number used as logic	17577 out of 69120	25%
Number used as Memory	601 out of 17920	3%
Number of External IOBs	4 out of 640	1%
Number of BUFGs	3 out of 32	9%

4. Conclusion and Future work

In this paper, we introduced SOA concepts to MPSoC platform for chip level parallelization. SOA concepts bring new opportunities to traditional MPSoC challenges including unified programming interfaces, automatic task partitioning, and out-of-order scheduling scalable hardware reconfigurations. Empirical results on FPGA shows that SOA can efficiently facilitate researchers to construct application specific MPSoC with adoptable modules with high flexibility. Both the software scheduling overheads and hardware utilization are acceptable.

This work is still working-in-progress and there are numerous future directions worth pursuing. First, task partition and adaptive mapping methodologies will be essential to efficiently support automatic parallelization. Second, although significant researches are underway for programming, little work has been done on integrating reconfigurable FPGAs with conventional programming paradigms. Finally, the chip-level parallelization also proposes new challenges on out-of-order task scheduling, to break the area and resources limitations of FPGA devices.

References

[1]. Shekhar, B. and Andrew A.C. 2011. The future of microprocessors. *Communications of ACM*, 54, 5(May. 2011), ACM, New York, NY, 67-77. DOI= http://doi.acm.org/10.1145/1941487.1941507

[2]. Satnam, S. 2011. Computing without processors. *Communications of ACM*, 54, 8 (Aug. 2011), ACM, New York, NY, 46-54. DOI=http://doi.acm.org/10.1145/1978542.1978558

[3]. David W., Anant A., Factored operating systems (fos): the case for a scalable operating system for multicores. 2009, *ACM SIGOPS Operating Systems Review*, 43, 2 (Apr.2009), ACM, New York, NY, 76-85. DOI=http://dx.doi.org/10.1145/1531793.1531805

[4]. Wang, C., Zhang J., et al. (2011). SOMP: Service-Oriented Multi Processors. *Proceedings of the 2011 IEEE International Conference on Services Computing*, IEEE Computer Society: 709-716. DOI=http://dx.doi.org/10.1109/SCC.2011.26

[5]. Etsion Y., Cabarcas F., Rico A., Ramirez A., et al., Task Superscalar: An Out-of-Order Task Pipeline, *in Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture.* 2010, IEEE Computer Society. p. 89-100. DOI=http://dx.doi.org/10.1109/MICRO.2010.13