Optimization of a Retargetable Functional Simulator for Embedded Processors

Francesco Papariello
Advanced System Technology
STMicroelectronics
Grenoble, France
francesco.papariello@st.com

Gabriele Luculli
Advanced System Technology
STMicroelectronics
Grenoble, France
gabriele.luculli@st.com

Abstract

The objective of this research is to develop tools and methods for system-level optimization of embedded software that is executed on system-on-chip platforms. In particular, this paper proposes a new instruction-set simulator's tool flow which has been extended with some retargetability features and multiple program representations. We propose an innovative way to improve the simulation speed, that is a key factor for embedded software optimization, by using a synthesis approach. In depth description is reported of the source-level optimization of the simulation library which is an important part of such new mechanism. Experimental results show a speedup of about 24 for the throughput of the simulation library, reaching the valuable performance of 50 Mops.

1 Introduction

Today System-on-Chip (SoC) designs of embedded applications are often built around one or more core processors that are mainly used as soft computing resources. High-performance computations are usually carried out on custom processor accelerators or on the core itself by enhancing its instruction set. Such platform configuration is certainly valid provided that the power consumption and the performance requirements are met. Moreover, this platform can easily satisfy all the other system-level requirements that are commonly found in a product definition. To name a few, flexibility for adaptation to new standards, easy add-on of new functionalities for a specific customer and fast upgrade to derivative products are some of them.

The design of software applications for this kind of hardware platform is based on the extensive use of *instruction set simulators* (ISS). Since embedded code requires tight optimizations to fully exploit the available hardware features, and the development boards are available very late in the design process, the use of complex ISS with detailed modeling of the hardware platform is the only viable solution. In the past, it was sufficient the use of bare ISS where just the simulation of the instruction set architecture (ISA) was considered in detail at the functional level and sometimes at micro-architecture level for performance estimation. Nowadays, new demanding needs are faced due to the difficulties of designing embedded software for core processors which are part of complex SoC, often with application-specific and heterogeneous memory architectures, complex register files and very specific communication mechanisms to cope with the presence of the many hardware accelerators. Moreover, the ISS are also used in later phases of the system design, e.g. they are an important part of hardware/software co–simulation [8], and sometimes, when the core processor requires an ISA extension, they are also used to validate the processor design.

Overall, such needs ask for a re-thinking of the ISS, shifting it from its basic role of ISA simulator for a standalone processor towards the one of simulator for a sub–system with many application-specific architectural and micro-architectural features. Obviously, this new "ISS for sub-systems design" should provide the required functionalities in an effective way, so the historical tradeoff between simulation speed and accuracy of the results is more than ever present. But, if it was important to have a high simulation speed in the past, it is paramount today in order to enable the exploration of more complex design spaces. A careful identification of the simulation bottlenecks is just the first step in the development of this new tool, following the use of extensive optimization techniques and new simulation semantics should be adopted.

In this paper we present the software architecture of an in–house functional simulator of STMicroelectronics and the optimization strategy that we have applied to it in order to improve its performance. The overall target use of our simulator is not only the design and validation of embedded software, but also to support the development of new optimization techniques which exploit the specific features of the hardware platform to improve the quality of the embedded code. For this reason, its structure is more complex and more general than a standard ISS, and its development raises new challenges. In this paper we describe its overall structure and the aspects related to the optimization of the

Proceedings of the Ninth Annual IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'02)
0-7695-1549-5/02 $17.00 © 2002 IEEE

IEEE COMPUTER SOCIETY

functional simulation, with a special focus on the optimization of its basic simulation library. The proposed results are innovative at least for:

- The new proposed schema for system-level simulation of sub-systems which is based on a simulation library of ISA-independent optimized operations, ISA-specific generated code to efficiently customize the simulator and several program representations.
- The methodology adopted to balance the local optimization of the simulation library, still targeting the global simulation performance.
- The development of a high-performance simulation library which reaches the remarkable throughput of 50 Mops in a 450 Mhz host machine.

This paper is organized as follows. In Section 2, we report the related research works. In Section 3, we describe the overall structure of the simulator, its simulation library and the adopted optimization strategy. In Section 4, we report the experimental results. Conclusions and future work appear in Section 5.

2 Background

There is a large literature in ISA functional simulation for both general–purpose and embedded processors. The proposed approaches span from fully *interpretive simulation*, where the basic simulation steps of target instruction decoding, decomposition in *basic operations* and simulation loop unfolding are executed at simulation time, up to fully *compiled simulation* where the same steps are executed at compilation time. In the middle there are many partially-interpretive and partially-compiled simulation approaches which try to exploit different static information from the ISA and dynamic information from the application to be simulated, in order to reach fast enough simulation speed with the required accuracy of the results [1]. When the simulation includes performance or timing, it is good practice to decompose the overall implementation in a functional model, which covers just the functional simulation of the ISA, and a performance model which has as main responsibilities the correct timed execution of the instructions and the collecting of the performance figures [2].

In any case the simulation framework is built around a *software library* which provides the basic functionalities for the simulation: the main loop and some glue functions to control the simulation are obviously part of it, but the larger part consists of implementation of some *basic operations*. Depending on the case such basic operations are coded in C language, to be compiled on the host machine, or they are directly coded in the host assembler to reach higher simulation speeds. It is very important to have a well structured set of basic operations, in order to assure an efficient decomposition of the target instruction, and a very efficient implementation of the simulation library. *Retargetability* of the simulation framework for different processors is an additional feature which is becoming more and more common in the industry. Even in this case a basic simulation library exists, but the achievement of high simulation speeds is much more difficult than the general case since the library should be optimized for many different ISA rather than for a single one.

In the following we describe the related research. A particular emphasis is made on the use of some basic software library for the coding of the ISA behaviours and its impact on the simulation performance. Even if this is an important subject for the efficient implementation of industrial simulators, we find out that few authors treated it in some details and, unfortunately, none in a comprehensive way.

2.1 Related Works

The approach proposed by Pees et al. [1] makes use of LISA machine description language to add retargeting capabilities to their simulation framework. From a LISA processor description they derive a C simulation library which is specific for the target processor but general enough to be used as common library for each application. From the same LISA model they also derive a processor-specific simulation compiler which will compile each application to a compiled simulator using the previous library as basic simulation resource. Neither they report any information about the structure of the used software library nor detail its performance figures, but it is clear that the high speed of their simulation experiments is at least partially due to an extremely efficient implementation of such library. How the processor-specific library is optimized, it is not described.

A different method was followed by Zhu and Gajski [3] which developed a retargetable compiled simulator by exploiting the availability of a retargetable compiler. The approach is quite interesting because it takes advantage of a virtual instruction set, used as intermediate representation of the embedded code, to derive either an assembly or a C simulation code for the host machine. They make use of an hand-written software library where each virtual instruction is implemented using host machine instructions. It is worth noting the way the virtual instructions are implemented sometimes has a clear influence on the simulation speed, but no indications are reported on adopted optimizations for such library. Their approach is portable, since the common virtual instruction set allows it, but it is in some way limited by the fact that the coding of a new library is required for each host machine. This is a time consuming and error prone activity.

In the paper [4] Chandra et al. describe a retargetable functional simulator which is based on the Sim-nML machine description language. The simulator takes binary code in ELF format as main input, and generates a C simulator. It makes use of a library of basic operations, which is common to any generated simulator, and a table of func-

Proceedings of the Ninth Annual IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'02)
0-7695-1549-5/02 $17.00 © 2002 IEEE
IEEE COMPUTER SOCIETY

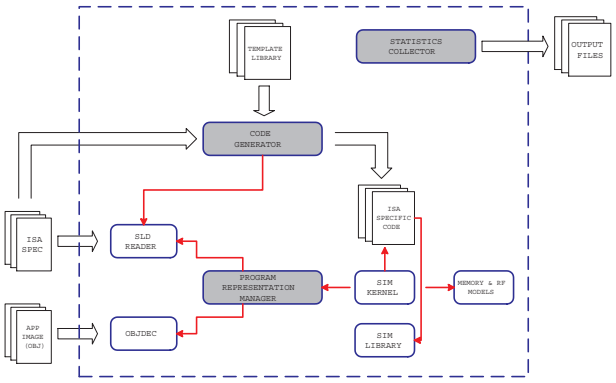

Figure 1. Flow diagram of the simulator

tionalities, which is specific for each target embedded code (i.e. one table entry for each target instruction in the binary code). An interesting conclusion which comes out from their experiments is that the average simulation speed is strongly dependent on the specific coding of the ISA behaviors and its variance is quite high. They don't report any motivation for these effects, but we guess they could be probably related to the use of a library that is not well balanced for retargetability or just partially optimized for performance.

3 Simulator Development and Optimization

The overall structure of our functional simulator is depicted in Figure 1. It takes two files as main inputs, the embedded application image and the ISA specification, and it produces any collected statistics or traces about the executed target instructions. The ISA specification is written in a subset of the SLED machine description language [5], and it is used to enable the retargetability of the tool. Several blocks which are present in the flow diagram are common to any ISS and they are implemented in a quite standard way:

- *ObjDec*: its main duty is to decompose an application image in a set of data and code sections, and to get the memory allocation information for each section. Actually, we are just able to read the ELF format.

- *Sim Kernel*: it is the main loop of the simulator, its unfolding drives the execution of the overall simulation.

- *Memory and RF Models*: they are the functional models of the memory structure and the register file. In the actual implementation this modules just provide the basic interface functionalities of data read/wite without any modeling of neither the microarchitecture structures nor the performance effects; in the future we plan to extend them to more realistic memory architectures and register file structures.

Comparing our flow diagram to other ISS we can note the presence of additional blocks. They are mainly due to the wider scope of our tool. Since we target sub-systems design which are built around one or more core processors, it is important to optimize the sub-systems in the overall, rather than their single components. In particular the quality metrics (e.g. performance and power consumption) of software applications obviously depend on the core characteristics, but much more on the overall interaction of the core itself with the memory architecture, the bus structure and the available hardware accelerators. Tight optimizations are practically achieved by designers through a strong matching of the software implementation with the hardware architecture. In order to automatize such process an abstract representation of the software application is clearly required. Neither this is the actual view that standard ISS have of the image code (because they make use of a very low-level view without any code structure representation). Nor this is the view that compilers have of the source code (because they are much more oriented to source-to-executable code translation rather than system-level design optimization). New specific program representations are required for the optimization of software that is embedded in a sub-system rather than a single core processor. Finally, we need also to improve the tool flow for higher simulation speeds in order to cope with its extended scope.

The additional blocks in Figure 1 are:

- *SLD Reader*: its main functionalities are to read and parse a SLED specification, then to build an internal representation of the target ISA which will be eventually used to decode any valid bit stream of code. Instructions decoding, like in compiled simulation, is done just one time for each target application so it is not in the critical path of the simulation.

- *Program Representation Manager*: standard ISS make use of a flat representation of the application image. For an interpretative simulator this is just the target image which is loaded by the memory module in a buffer in the host machine. In contrast for a compiled simulator, the application image is represented by a dispatch table which associates to each target instruction of the image one or more calls to the functions which simulate such instruction (e.g. an example is reported in [1]). In the actual implementation of our tool, we adopted just a flat representation since this was sufficient to validate the overall tool flow but, as our research will progress, additional representations will be added in order to develop new sub-system optimizations.

- *Code Generator*: retargetable simulators usually have some kind of code generation from an input spec, but in general this is implemented as one-to-one translation from the spec constructs to C function calls. We believe that this simple translation process is not sufficient to provide high simulation speeds. In fact, there

Proceedings of the Ninth Annual IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'02)
0-7695-1549-5/02 $17.00 © 2002 IEEE
IEEE COMPUTER SOCIETY

is a big potential improvement by adopting a synthesis approach (like the one used in digital design [6]) that builds up an optimized implementation of the spec constructs by exploiting an available library of basic operations. In the current version of the simulator the path from ISA spec to ISA specific code is not automatized yet. Actually the requirements for the code generator and for the structure of the template library are part of the research results reported in this paper.

- *Sim Library*: it is the library which implements the basic operations in which each target instruction is decomposed. Since this is the lowest level in the simulation path (started from the Sim Kernel module through the ISA Specific Code and down to it), the overall simulation speed is limited by the efficiency of this library. Extremely short execution time of the operations implemented in the library is our first requirement. Moreover the tool environment should be retargetable, so we have an additional requirement which is to select a set of basic operations that are general enough to implement any possible ISA for embedded processors. This second need adds a lot of complexity to the development of the library because the grain size of its operations has contrasting impact on the library performance and the overall simulation performance. In fact the library performance is independent from the target ISA, i.e. the variability of the ISA has no impact on it, while the overall simulation performance depends on both the specific ISA and the library performance. Implementing the library as a small set of fine-grain operations is very time efficient, but each target instruction will be probably decomposed into many of such operations. Otherwise, if the library is implemented as a larger set of medium-grain operations, the library will have a lower performance and each target instruction will be decomposed into a smaller number of basic operations. Obviously this second approach is less prone to tool's retargetability because each basic operation is much more specific. Which is the best tradeoff between the operation's grain size and the number of basic operations required to implement the ISA behaviours in order to maximize the simulation performance, and assuming a variable ISA, is a key problem to solve. We propose a compound solution based on a strong optimization of the Sim Library, for reaching the highest performance for each basic operation, and on automatic code generation in order to exploit all the possibilities of reducing the number of executed basic operations for each target instruction. As far as we know, nobody has previously tried this interesting solution in the field of retargetable ISS.

The rest of this paper describes the general structure of the Sim Library and the details of the optimizations we performed on it.

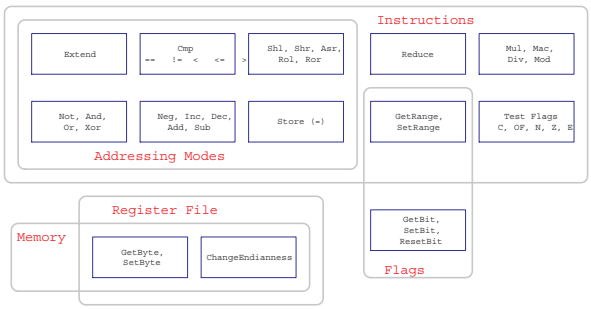

Figure 2. Composition of the Sim Library

3.1 Structure of the Sim Library

The set of basic operations which compose the Sim Library are depicted in Figure 2, where they are grouped in several functionality classes according to their use in the simulator. In general, the simulation of an instruction goes through all the four steps: evaluation of the addressing mode for each operand, access to the operands in memory or in the register file, execution of the specific computation and write back of the results in the register file, memory or control flags. At least one operation from each class is always evaluated for each instruction and most of the time many of them are required to simulate a single instruction, so it is clear that the performance of the simulation is strongly dependent on the efficient implementation of the overall set.

A useful way of representing the breakdown of the simulation time is reported in Figure 3, where the functionality classes are shown with their percentage impact on the simulation time and the use-of relation between functionalities is showed. The bottom of the hierarchy is filled by our Sim Library and, as expected, it is the block with the highest impact on the total simulation time, i.e. 85.17 %. It is also evident from the picture that the self execution time of each class is always extremely small, that is the total execution time of each class is mainly due to the use-of relation of some other class, rather than to the functionalities implemented in the class itself, and in the end to the use of the Sim Library. This is clearly a bottleneck for the simulation and in fact the average library's performance originally was quite low, about 1-2 Mops, which means that if we assume that 20 to 50 basic operations are needed to simulate a target instruction, we could reach a poor simulation performance of 20-100 Kinstr/sec.

So, it is important to reduce the dependency of the simulation time on the Sim Library, but it is worth noting that in general such kind of strong dependency is not a disadvantage by itself. In fact it could be acceptable for a custom functional simulator where the overall functionalities are provided by the library and the other classes (e.g. memory, flags, etc) should implement just some useful interfaces. However, it becomes a clear drawback when we consider an ISA–retargetable functional simulator which, moreover,

Proceedings of the Ninth Annual IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'02)
0-7695-1549-5/02 $17.00 © 2002 IEEE
IEEE COMPUTER SOCIETY

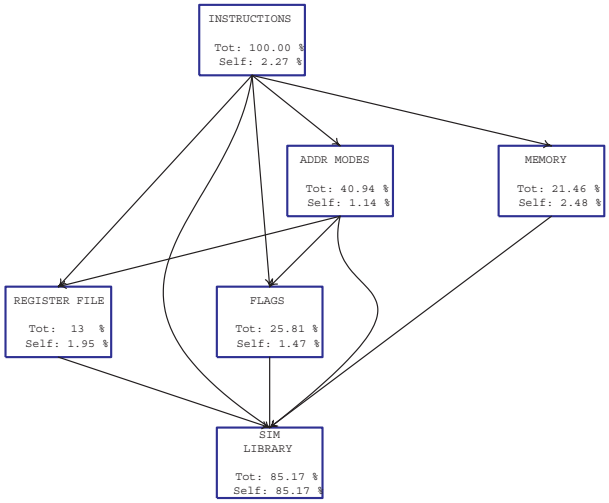

Figure 3. Reduced call graph (without optimizations)

aims to extend its retargetability features to a complete subsystem. A coarse-grain library exceptionally tuned for a given ISA performs well on the specific case, but probably quite bad on all the others. So, a careful tuning of its functionalities grain-size and the ISA-independent optimization of them are major goals to develop effective retargetable simulation.

3.2 Source Code Optimization Techniques

In order to optimize the Sim Library we have adopted several techniques, some of them have been globally applied to the overall library others have been locally applied to the single operations.

Our first step in optimizing the library has been the modification of the used data structures and in particular of the representation of bit sequences, which are the main operands of any library operation. Originally, we used dynamic allocation for any bit sequence and an internal representation based on a concatenation of byte units, which did not match well with the typical word length of the host machine. So, a lot of time was wasted to allocate and deallocate memory on the host heap during the simulation and a large overhead was present to compose any basic arithmetic/logic operation on bit sequences in terms of operations on host words. Moreover some operations, which are frequently used in the memory module (i.e. GetByte, SetByte and ChangeEndianess), usually need to access the bit sequence as a sequence of bytes. The efficient execution of such operations strictly depends on the endianess of the host machine but originally we did not put any attention in their implementation, so a lot of simulation time was wasted to find out which byte to access instead of accessing it. Similar flaws were present in the overall software architecture of the Sim Library: the general interface of several functionalities was too flexible, allowing the execution of the function on bit sequences of any bit lengths, and on any couple of bit sequences without any restriction. The many functionalities in Figure 2 had inter-related implementations, so the execution of just one of them implied the execution of many others, resulting in a library with a quite deep hierarchy. In order to overcome these problems, we have applied the following global changes to the library:

- Now the internal representation of bit sequences is still based on concatenation of some basic units, but the unit's size is customizable (it is usually chosen to match the host word size) and the sequence's endianess is kept the same as the host memory. In addition, all the data structures are statically allocated.

- The operation's interfaces are much more constrained, just allowing the use of bits operands with the same size which is usually defined at compile time.

- The hierarchical implementation of the library has been flattened for most of its functions, and the dependencies between functions have been removed as much as possible. Few operations (i.e. Div, Mod), which don't allow a simpler implementation, still require multiple calls to some other operations, so the overall hierarchy is not completely flat.

Following the global restructuring, several optimizations have been locally performed to each operation. Basically, we have used three different techniques: code specialization, function specialization and C/ASM matching. Code specialization tries to improve the performance of source code by specializing it for frequent expected situations. The specialized code performs the same computation as the original one, but with reduced computational effort by exploiting the knowledge of constant values for some variables [7]. Function specialization is conceptually similar to code specialization, but here the specialization acts on the functionality itself, instead of on the code, which is specialized for some specific conditions. C/ASM matching is the iterative process of adapting the C source code in order to reduce the number of assembler instructions into which it is translated. So it is the practical attempt to write host-independent C code, but thinking in terms of the commonly available assembler instructions, reducing the number of read-write dependencies by rewriting/swapping expressions in the source code, and trying to reduce the effects of register spills through a careful use of variables.

Each technique is more or less effective depending on the specific case, moreover they can be jointly applied to the same computation in order to enhance the overall optimization. Code specialization has the best impact on computations which have a lot of static information to exploit and where the number of expected situations is quite small, so

Proceedings of the Ninth Annual IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'02)
0-7695-1549-5/02 $17.00 © 2002 IEEE
IEEE COMPUTER SOCIETY

Level vs. Technique	Code specialization	Function specialization	C/ASM matching	Global restructuring
Fully optimized	GetBit, SetBit, ResetBit (1.5, 1.3)		GetByte, SetByte, ChangeEndianess (1.7, 1.5)	Data structures Functions interfaces (3.9, 2.4)
Partially optimized	Add, Sub Cmp, <=, >=, ==, != (1.3, 1.1)	R/W in Memory R/W in Register File (1.9, 1.7)		
Don't need	- Not, Or, And, Xor, Store; Test Flags			
Not possible	- Extend, Reduce; - Shl, Shr, Asr, Rol, Ror; - SetRange, GetRange; - Mul, Mac, Div, Mod; - Add, Sub.			

Figure 4. Classification of the Sim Library's operations by optimization level and optimization technique (library and simulation speedups in the brackets)

Benchmark	Description
`matcnt`	Summation of two 100x100 matrices
`stats`	Calculate sum, mean and variance of two arrays
`whetston`	Whetstone Benchmarks
`fft`	Cooley-Tukey Fast Fourier Transform on 1024 points
`jfdctint`	JPEG slow-but-accurate implementatin of DCT

Table 1. Benchmarks description

that few optimized computations keep the place of a complex one. However, if static information are not available or if it is difficult to identify useful expected situations, it is always possible to apply function specialization in order to customize a functionality for a specific purpose. This adds some complexity to the library, but it pays off if the added flexibility is used in a functionality which has a broad impact in the simulation. Finally, C/ASM matching can always be applied but it requires the use of suitable data structures in the C source code which fit the actual implementation at the assembler level and memory organization of the host machine. In the following section, the optimizations applied to the Sim Library are described in details and the gains on the simulation speed are reported.

4 Experimental Results

We have fully optimized several basic operations reaching the highest possible performance, for others we have just achieved a partial optimization since it has not been possible to reduce further their execution time just working at the source code level and for some others it has not been possible to apply any optimization at all. Obviously, there are also some basic operations which do not need to be optimized. Figure 4 shows a classification of our library's operations in terms of achieved optimization levels and applied optimization techniques. The numbers in round brackets are respectively the average library and simulation speedups. We have used the same host/target configuration for all the experiments: a 450Mhz Linux host machine, the ISA spec of the ARM7 embedded processor and a small set of benchmarks (Table 1).

Code specialization has been successfully applied to the *GetBit*, *SetBit* and *ResetBit* operations, resulting in an average library speedup of 1.5 and an average simulation speedup of 1.3. They were prone to be specialized because the number of expected situations for them is small (i.e. it is equal to the maximum number of bits in a typical sequence) and most of their computational efforts were related to mask and shift operations, which can be easily evaluated off-line whereas they are quite time consuming at run-time. The original and the optimized versions of GetBit are reported in Figure 5. It is quite evident the effectiveness of code specialization: actually the main body of the optimized code is just six assembler instructions that is the strictly minimum which is required to access to an element of the bits sequence first and then to mask it in order to get a single bit. A similar strong reduction of the computational effort has been obtained for the SetBit and ResetBit basic operations.

Other basic operations, e.g. several kinds of compare functions of bit sequences, are not prone to be fully specialized, since they have a large set of expected situations. Anyway, we have applied partial code specialization to some boundary situations which, even if they are not the most frequent conditions in absolute, are part of the most time consuming ones. We have gained a library speedup of 1.3 and a simulation speedup of 1.1.

We have applied function specialization to the *GetRange* and *SetRange* basic operations, resulting in an average library speedup of 1.9 and an average simulation speedup of 1.7. These functions are used inside the memory and register file modules (e.g. by the Read_16 procedure in Figure 6), so they are called many times during the simulation. We have specialized their functionalities into two new procedures, respectively GetByte and SetByte, which just get or set a byte of a bits sequence, instead of getting or setting a general sub-sequence of bits as the original functions did. The GetByte and SetByte functions are now part of the Sim Library, and they are mainly used by the memory and register file modules. Whereas the GetRange and SetRange are still used to get/set the status flags. Figure 6 reports the old and new implementations of an interface procedure of the memory module where now the SetByte is used. Even

Proceedings of the Ninth Annual IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'02)
0-7695-1549-5/02 $17.00 © 2002 IEEE
IEEE COMPUTER SOCIETY

```
bool GetBit(n) {
  return (seq[n/BBD] & (1<<(n%BBD)))!=0;
}
```

```
bool GetBit(n) {
  return (seq[tab1[n]] & tab2[n])!=0;
}
```

```
UINT8 GetByte(pos) {
  return (UINT8)(seq[last_elem pos/BBB]
                 >> ((pos%BBB)*8));
}
```

```
UINT8 GetByte(pos) {
  return *((UINT8*)seq+pos);
}
```

GetBit in ASM:

```
    pushl %ebp
    movl %esp,%ebp

    subl $4,%esp
    pushl %ebx
    movl 12(%ebp),%eax
    movl %eax,%ebx
    movl %eax,%edx
    testw %ax,%ax
    jge .L
    movl %ebx,%edx
    addl $31,%edx
.L: movl %edx,%eax
    sarw $5,%ax
    movswl %ax,%ecx
    movl %ecx,-4(%ebp)
    salw $5,%ax
    subl %eax,%ebx
    movswl %bx,%edx
    movl $1,%eax
    movl %edx,%ecx
    sall %cl,%eax
    movl 8(%ebp),%edx
    movl 4(%ebp),%ecx
    testl %eax,(%edx,%ecx,4)
    setne %al
    movl 8(%ebp),%ebx

    leave
    ret
```

GetBit in ASM:

```
    pushl %ebp
    movl %esp,%ebp

    movl 8(%ebp),%ecx
    movswl 12(%ebp),%edx
    movsbl tab1(%edx),%eax
    movl (%ecx,%eax,4),%eax
    andl tab2(,%edx,4),%eax
    setne %al

    leave
    ret
```

GetByte in ASM:

```
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx

    movl 8(%ebp),%edx
    movl 12(%ebp),%ecx
    movl 8(%edx),%ebx
    decl %ebx
    movl %ecx,%eax
    testl %ecx,%ecx
    jge .L
    leal 3(%ecx),%eax
.L: sarl $2,%eax
    subl %eax,%ebx
    movl (%edx),%edx
    sall $2,%eax
    subl %eax,%ecx
    sall $3,%ecx
    movl (%edx,%ebx,4),%eax
    shrl %cl,%eax
    testl %eax,(%edx,%ecx,4)
    andl $255,%eax
    movl 4(%ebp),%ebx

    leave
    ret
```

GetByte in ASM:

```
    pushl %ebp
    movl %esp,%ebp

    movl 8(%ebp),%ecx
    movsbl 12(%ebp),%eax
    movzbl (%eax,%edx),%eax

    leave
    ret
```

Figure 5. Implementations of GetBit and GetByte functions, respectively, before and after optimization

if this is a small procedure it is quite evident the double source of improvement, gained by function specialization: its main body has a simpler structure, resulting in a reduced computational effort, and it is implemented by very efficient dedicated functions.

An interesting thing about function specialization is that it can bring new opportunities for optimization, so globally enhancing its effect on the performance improvement. The specialization of GetRange and SetRange is such case, in fact the new added basic operations were prone to be optimized through C/ASM matching. As reported in Figure 4 their optimization, together with *ChangeEndianess*, results in a very good speedup. Figure 5 shows the original and new implementation of the GetByte function.

Taking into account the overall optimizations the Sim Library gains a huge speedup of about 24.5, which results in a final library's performance of 10-50 Mops. The top value is obviously related to the optimized basic operations, whereas the lower bound depends on the set of basic operations which we have not optimized yet because the usually used optimizations didn't apply to them. This final result is certainly valuable, in fact a 50 Mops throughput implies that a simulation performance of several Mips should be reachable by coding each target instruction with just 20-50 basic operations (which was our initial requirement). Unfortunately, the variance of the library's performance needs to be reduced in order to guarantee a reliable simulation performance, which should be quite stable independently from the considered input target image and ISA spec.

Considering the average simulation speed for the benchmark set, the use of the optimized library results in a simulation speedup of 8.7. This is, as expected, lower than library speedup since the execution time of the library is just a fraction of overall execution time. In Figure 7, it is reported the new breakdown of the simulation time. Comparing the new self and total figures with the old ones (Figure 3) we can note that:

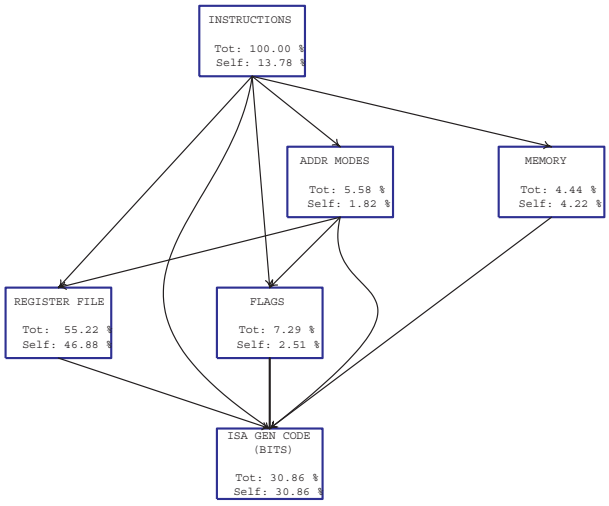

Figure 7. Reduced call graph (with library optimizations)

Proceedings of the Ninth Annual IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'02)
0-7695-1549-5/02 $17.00 © 2002 IEEE

IEEE COMPUTER SOCIETY

```
Bits &Read_16(addr) {
  static Bits tmp16(16), tmp8(8);

  tmp8=_Read(addr);
  tmp16.SetRange(0,tmp8);
  tmp8=_Read(addr+1);
  tmp16.SetRange(8,tmp8);

  return tmp16;
}
```

```
Bits &Read_16(addr) {
  static Bits tmp16(16);

  tmp.SetByte(0,_Read(addr));
  tmp.SetByte(1,_Read(addr+1));

  return tmp16;
}
```

Figure 6. Implementations of the Read_16 function for the memory module before and after function specialization of the SetRange into SetByte

- The Sim Library is no more the main simulation bottleneck, in fact its self execution time is now just a third of the total one, whereas originally it was as much as the 85%.
- Side-effect results of the library optimization are the general reduction of dependency that the other modules have from the library, and a general increasing of the self execution time. These can be reasonably understood as an improved localization of the simulator functionalities to the logical module where they are actually implemented. That is, simulation time is used exactly where it is required and when the called functionality is useful for the simulation. The overall distribution of the total and self execution times shows that, thanks to the accurate selection of the set of library's basic operations and their grain size, the software architecture of the simulator is now well balanced.
- The register file module is clearly the new bottleneck with a self execution time of about 47%.

In summary, the applied optimizations to the simulation library are proved to be very effective in improving its throughput and to have a substantial impact in the overall simulation performance. The use of multiple optimization techniques, in order to exploit the available characteristics of the basic operations, has been essential to reduce their computational effort for most of them. However, for others it has not been possible to gain any useful reduction through source-code optimizations (i.e. the "not possible" class in Figure 4). In the future we plan to explore a synthesis approach in order to reduce their effort as well. The main requirement, which might restrict in some way the usefulness of the synthesis for this problem, is to be able to upgrade their performance of a factor five, at least, to derive a full library with a stable performance of 50 Mops. Such challenging objective is the topic of future research.

5 Conclusions and Future Work

We have presented an extension of a standard ISS tool flow for system-level design and optimization of embedded software which is executed on complex sub-systems. A synthesis-approach has been proposed as innovative solution to the today present dichotomy between the needs of higher simulation speeds on one side, and simulation of more and more complex functional models on the other side. The extensive optimization of the simulation library, which is a ground component for any synthesis approach, has been carried out, and its impact in the overall simulation time has been carefully evaluated. In the future, we plan to develop the synthesis path of our tool flow and to add some structural representations of the application, in order to build automatic optimization algorithms for embedded software.

References

[1] S. Pees, A. Hoffmann, and H. Meyr. Retargetable compiled simulation of embedded processors using a machine description language. *ACM Transactions on Design Automation of Electronic Systems.*, 5(4):815–834, Jan. 2000.

[2] P. Bose and T. M. Conte. Performance analysis and its impact on design. *IEEE Computer*, 31(5), 1998.

[3] J. Zhu and D. D. Gajski. A retargetable, ultra–fast instruction set simulator. In *Proceedings of the 1999 Conference on Design, Automation and Test in Europe (DATE-99)*, Los Alamitos, CA, 1999. ACM/IEEE.

[4] S. Chandra and R. Moona. Retargetable functional simulator using high level processor models. In *Proceedings of the 13th International Conference on VLSI Design*. IEEE, 2000.

[5] N. Ramsey and M. F. Fernández. Specifying representations of machine instructions. *ACM Transactions on Programming Languages and Systems*, 19(3):492–524, May 1997.

[6] G. D. Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw–Hill Publisher, 1994.

[7] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

[8] K. Keutzer. Hardware/software co-simulation. In *Proceedings of the 31st Conference on Design Automation*. ACM Press, June 1994.

Proceedings of the Ninth Annual IEEE International Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'02)
0-7695-1549-5/02 $17.00 © 2002 **IEEE**

IEEE COMPUTER SOCIETY