TotalProf: A Fast and Accurate Retargetable Source Code Profiler

Lei Gao, Jia Huang, Jianjiang Ceng,
Rainer Leupers, Gerd Ascheid, and Heinrich Meyr
Institute for Integrated Signal Processing Systems
RWTH Aachen University, Germany
{gao,leupers}@iss.rwth-aachen.de

ABSTRACT

Profilers play an important role in software/hardware design, optimization, and verification. Various approaches have been proposed to implement profilers. The most widespread approach adopted in the embedded domain is *Instruction Set Simulation* (ISS) based profiling, which provides uncompromised accuracy but limited execution speed. Source code profilers, on the contrary, are fast but less accurate. This paper introduces *TotalProf*, a fast and accurate source code cross profiler that estimates the performance of an application from three aspects: First, code optimization and a novel *virtual compiler backend* are employed to resemble the course of target compilation. Second, an optimistic static scheduler is introduced to estimate the behavior of the target processor's datapath. Last but not least, dynamic events, such as cache misses, bus contention and branch prediction failures, are simulated at runtime. With an abstract architecture description, the tool can be easily retargeted in a performance characteristics oriented way to estimate different processor architectures, including DSPs and VLIW machines. Multiple instances of *TotalProf* can be integrated with *SystemC* to support heterogeneous *Multi-Processor System-on-Chip* (MPSoC) profiling. With only about a 5 to 15% error rate introduced to the major performance metrics, such as cycle count, memory accesses and cache misses, a more than one *Giga-Instruction-Per-Second* (GIPS) execution speed is achieved.

Categories and Subject Descriptors

B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids

General Terms

Design, Measurement, Performance

Keywords

Source Code Profiling, Performance Estimation, Instruction Set Simulation, Architecture Description Language

1. INTRODUCTION

Profiling is deemed of pivotal importance for embedded system design. On the one hand, profiles obtained from realistic workloads are indispensable in application, processor, and compiler design. On the other hand, profile-based program optimization (e.g., [5]) and parallelization (e.g., [7]) play an increasingly important role in harnessing the processing power of embedded processors. However, current profiling techniques face two major challenges: First, the stringent time-to-market pressure and the increasing complexity of applications and systems require profiling tools to be fast and quickly available, so that they can be employed as early as possible in the evaluation of design variants. Second, the accuracy of profiles is imperative to capture the characteristics of today's highly diverse embedded applications and systems. Unfortunately, no existing profiling technique can meet all these requirements, as briefly outlined in the following:

- *Instruction Set Simulation* (ISS) based profiling is the most widespread approach in the embedded domain. Profiling is performed during simulation, therefore uncompromised accuracy is achievable. However, the execution speed is inadequate in many scenarios. Additionally, modeling and modifying simulators are nontrivial, which limit the early availability of the ISS-based profilers.

- Conversely, classic source code profilers encompassing *Source-Level Performance Estimation* (SLPE) [16, 13, 18] utilize machine-independent optimizations provided by the host compilers to resemble the optimizing target compilation process for the sake of accuracy. However, the accuracy is still prohibitively limited, especially for VLIW architectures and *Application Specific Instruction-set Processor*s (ASIPs), which are regrettably the main areas where accurate profiling is needed.

Figure 1 compares the workflow of source code profilers with SLPE, ISS-based profilers, and *TotalProf* – the source code profiling infrastructure proposed in this paper. *TotalProf* features a novel *Intermediate Representation* (IR)-to-IR transformation process called *virtual compiler backend* (or *virtual backend*) to resemble the behavior of a real target compiler backend. It also estimates the behavior of the target processor's datapath and simulates the dynamic events, such as cache misses, bus contention and branch prediction

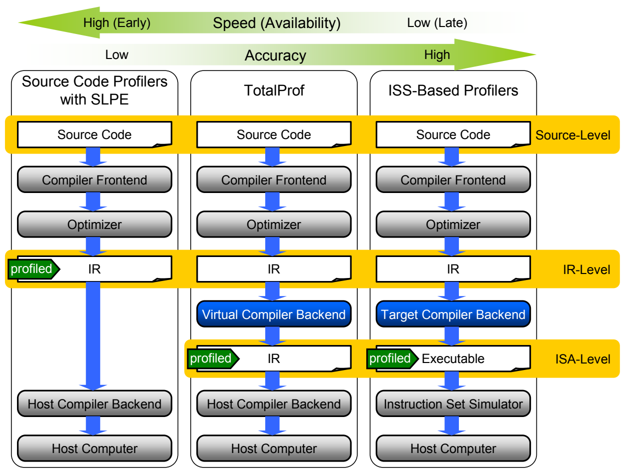


Figure 1: Concept of *TotalProf*. Like source code profilers utilizing SLPE, *TotalProf* uses host compilation to benefit from native execution. The profiling is performed at the same level of ISS-based profilers, therefore accuracy can be preserved.

failures. *TotalProf* can be easily retargeted with *architecture descriptions*. It can provide highly precise profiling information, which is normally only about 5 to 15% worse than the ISS-based profilers. At the same time, *TotalProf* executes at a more than one GIPS speed, which is 1 to 3 orders of magnitude higher than the speed of most ISSs. Furthermore, heterogeneous MPSoCs can be accurately estimated using multiple *TotalProf*s connected with *SystemC* bus/peripheral models.

The remainder of this paper is organized as follows. After reviewing the related work in section 2, section 3 elaborates the *TotalProf* infrastructure. Section 4 presents experimental results and two case studies on profile-aided VLIW architecture and MPSoC *Design Space Exploration* (DSE). After *TotalProf* is introduced, some in-depth discussions are given in section 5. Finally, section 6 concludes this paper.

2. RELATED WORK

2.1 Profiling Techniques

In general, profiling can be implemented using two means: instrumentation or supervision. Source code profilers are usually implemented using the former. They inject extra code to the applications' source code before or during compilation, so that profiles can be collected during the course of execution. As an example, GCC can inject extra profiling code to an application. After execution, a profile is generated, which can be analyzed and displayed using the *GNU gprof* [11]. This approach is intrusive in two aspects: First, some compiler optimizations (e.g., function inlining) might be suppressed due to the introduction of profiling code. Second, the timing information observed from the environment is also affected by the execution of profiling code. Moreover, it only generates profiles for the native environment and cannot provide the convenience of profiling an application for a target architecture on a host computer.

Source-level performance estimation that performs *cross profiling* can be utilized to address these issues. For example, the *micro-profiler*, proposed by Karuri et al. [16], utilizes SLPE for fine-grained source code profiling. It estimates the performance of an application by lowering the C source code to a so-called *3 Address Code* (3-AC) IR, which is executable C code that only consists of statements in a similar abstraction level to a RISC. After machine-independent optimizations are performed, the performance of each basic block in the 3-AC IR can be estimated. Together with the frequencies of basic blocks obtained from the execution, the performance of the entire application can be calculated.

Source code profilers only support a limited number of *High-Level Languages* (HLLs). The approach of instrumenting the applications' binaries [34, 24, 25, 21] overcomes this limitation. Recently, *Dynamic Binary Instrumentation* (DBI) tools, such as *Valgrind* [25] and PIN [21], become especially widespread in the general purpose domain. With these tools, application binaries are transformed and instrumented at runtime. However, developing a DBI tool for a new architecture is difficult and even not always possible. Because in the embedded domain the diversity of *Instruction-Set Architectures* (ISAs) is much higher than that of the HLLs, binary instrumentation based profiling is not widely used.

Supervision-based profiling can be implemented in various ways as well. The most straightforward example is hardware performance counters [1] that widely exist in modern processors. Advanced hardware profilers (e.g., [36]) can be used for complex cases, but the storage of the generated profiles is usually a bottleneck. Hardware profilers cannot be used in early design phases, when the prototypes are not available yet. A supervision-based profiler can also be implemented by modifying the software stack the applications run upon [17].

In the embedded domain, the most straightforward and widespread approach is ISS-based profiling, which performs profiling during simulation. For example, [6] employs *SystemC*-based co-simulation for profiling. Static information is collected using an ISS, and the result can be reused in dynamic event simulation.

Furthermore, both instrumentation and supervision based profiling approaches can benefit from sampling profiling [24, 14] that reduces the runtime overhead.

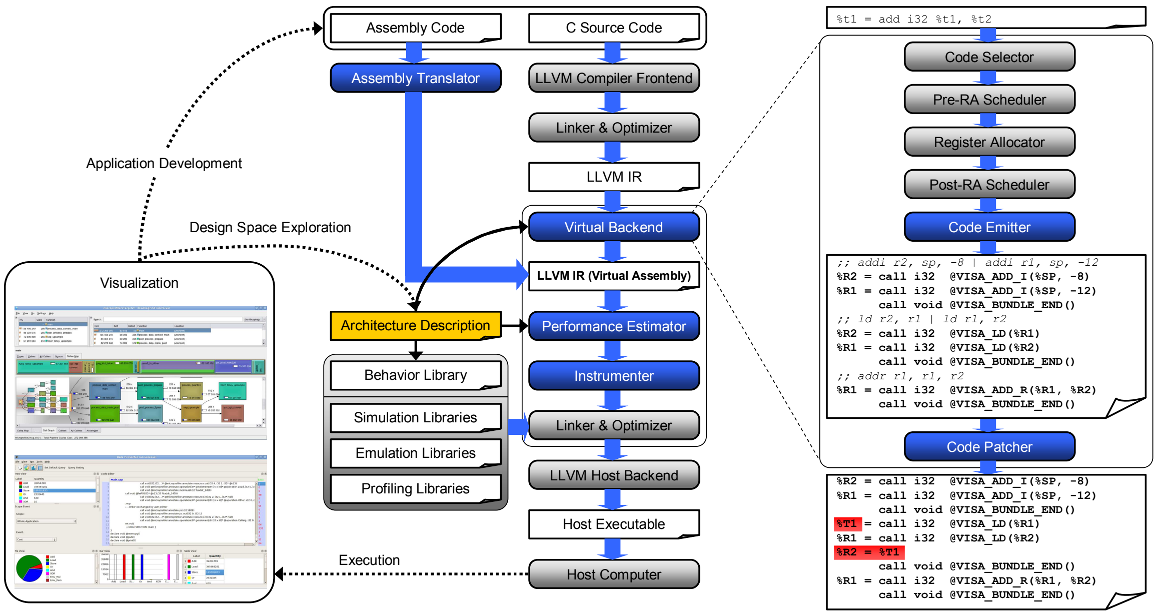
2.2 Instruction Set Simulation

Instruction set simulators are widely used in the embedded domain for profiling purposes.

SimpleScalar [4] is an interpretive simulator that can model a wide range of architecture/micro-architecture features. However, the execution speed is low since the entire process including instruction fetching, decoding and execution has to be simulated at runtime.

Compiled simulation is proposed to alleviate the execution effort by pre-decoding instructions. The decoding can be performed statically [38] or dynamically [26, 30, 29]. The latter, also known as *Just-In-Time Cached Compiled* (JIT-CC) [26] or IS-CS [30] simulation, can support self-referential and self-modifying code. Compiled simulators normally execute at several to tens of MIPS, which is much higher than the speed of interpretive simulators.

Binary translation is also used for simulation purposes [37, 2, 15]. Target binaries are directly translated to host executable code, and even inter-basic-block optimizations can be performed to further improve the execution speed [15]. Up to several hundred MIPS can be achieved in such



```
%t1 = add i32 %t1, %t2
```

```
;; addi r2, sp, -8 | addi r1, sp, -12
%R2 = call i32  @VISA_ADD_I(%SP, -8)
%R1 = call i32  @VISA_ADD_I(%SP, -12)
      call void @VISA_BUNDLE_END()
;; ld r2, r1 | ld r1, r2
%R2 = call i32  @VISA_LD(%R1)
%R1 = call i32  @VISA_LD(%R2)
      call void @VISA_BUNDLE_END()
;; addr r1, r1, r2
%R1 = call i32  @VISA_ADD_R(%R1, %R2)
      call void @VISA_BUNDLE_END()
```

```
%R2 = call i32  @VISA_ADD_I(%SP, -8)
%R1 = call i32  @VISA_ADD_I(%SP, -12)
      call void @VISA_BUNDLE_END()
%T1 = call i32  @VISA_LD(%R1)
%R1 = call i32  @VISA_LD(%R2)
%R2 = %T1
      call void @VISA_BUNDLE_END()
%R1 = call i32  @VISA_ADD_R(%R1, %R2)
      call void @VISA_BUNDLE_END()
```

Figure 2: *TotalProf* infrastructure with an example of a virtual backend

simulators. However, they are not widely used for profiling purposes, since the accuracy is often sacrificed to achieve high execution speed.

An orthogonal direction of research is sampling/statistical simulation [33, 9], which relies on cycle-accurate simulators but aims at reducing the simulation workloads. These approaches are mainly used in micro-architecture design space exploration.

2.3 Timing-Annotated Native Execution

Instead of simulation, the source code of applications can be directly compiled and executed on the native host computers, in order to reach a higher execution speed. Timing information can be annotated to the source code to facilitate profiling.

One way of obtaining the timing information is to analyze the applications' source code. This approach is also known as source-level performance estimation. The aforementioned *micro-profiler* is an example. Similarly, [13] applies SLPE with optimistic static scheduling to better resemble the characteristics of target processors. Nevertheless, SLPE only provides limited accuracy to a narrow range of target processors, albeit early availability is granted since there is no dependency on the target simulators or compilers.

Instead of analyzing the timing information from the source code, [18, 8, 31] perform the entire code generation using modified versions of the target compilers. The modified target compilers emit C code instead of target machine assembly to enable native compilation. However, the effort of applying this approach is not less than retargeting a real compiler. As a comparison, *TotalProf* can be conceptually regarded as a similar work that is devoted to alleviate the effort of retargeting.

Decompiling target binaries to HLLs is also researched [20], however this approach is more like binary translation and the dependency to target compilers still occurs. There are also endeavors on analyzing the target binaries or target compilation processes and annotating the information back to the (original or transformed) C source code [22, 32, 3]. However, these approaches lack evaluation with complex target code generation scenarios (as they make the annotation significantly difficult) and depend on the completion of the target compilers as well.

It is also important to mention that profiling is usually more than performance estimation. For example, most performance estimation tools cannot generate a call graph, memory access histogram, or a data dependence graph. Note, that further discussion will be given in section 5 once *TotalProf* has been introduced.

3. TOTALPROF INFRASTRUCTURE

Different to the aforementioned prior works, *TotalProf* aims at three goals at once: (1) An ultra fast execution speed that is on a par with SLPE. (2) A performance estimation close to ISS for a wide spectrum of processor architectures. (3) Ease to retarget. To accomplish these goals, *TotalProf* employs changes to the classic structure of SLPE.

As shown in Figure 2, *TotalProf* performs source-level performance estimation that is implemented by embedding several IR-to-IR transformations into the host compilation workflow of an open-source retargetable compiler – LLVM [19]. LLVM does not emit machine assembly for each compilation unit (i.e., source file), but generates a *bitcode* IR. The IR files of all the computation units can be linked to an archived IR, on which interprocedural optimizations can be performed. Originally, the LLVM host backend produces machine assembly from the archived IR. Instead, in *TotalProf*, some IR-to-IR transformations are performed before the host backend takes place. These transformations consist of a virtual backend, a performance estimator and an instrumenter, which can be retargeted using *architecture descriptions*. Profiling is performed at runtime, and the results can be visualized to guide application development and architecture design space exploration. The following subsections describe the virtual backend, performance estimation technique, and how to practically retarget *TotalProf*.

3.1 Virtual Backend

Unlike a real compiler backend, a virtual backend emits special LLVM IR called *virtual assembly*. The virtual assembly represents target-ISA-level information. Therefore, the course of target code generation can be simulated by retargeting the virtual backend using architecture descriptions. As shown in the right-hand side of Figure 2, the example VLIW virtual backend performs code selection, *pre-Register-Allocation* (pre-RA) scheduling, register allocation, *post-Register-Allocation* (post-RA) scheduling, etc., which can be found in most of the modern retargetable compilers. By describing each instruction's syntax, virtual assembly can be generated from a code emitter. As shown in the example output, global variables are used to represent registers, and function calls (to the implemented behaviors of corresponding instructions) are used to represent instructions. They are called *virtual registers* and *virtual instructions* respectively. Finally, the end of each VLIW bundle is also explicitly given.

Nevertheless, even with the behavior of each instruction provided, the directly generated virtual assembly may still execute incorrectly. The reason is twofold. First, each virtual instruction must be sequentially executed as an LLVM IR, while a processor may issue instructions in parallel. Second, the execution result of each virtual instruction is immediately written back to the virtual registers or the memory, but a pipelined processor architecture without interlocking (e.g., the MIPS-4K) can issue instructions before the results of previous instructions are written back, in which case original values are still accessible. A code patcher is introduced to address this issue. It creates temporary variables to buffer the execution results that cannot be immediately written back, so that the generated virtual assembly can be executed correctly. As shown in the example, a temporary variable `T1` is employed to buffer the value that cannot be immediately written back to `R2`.

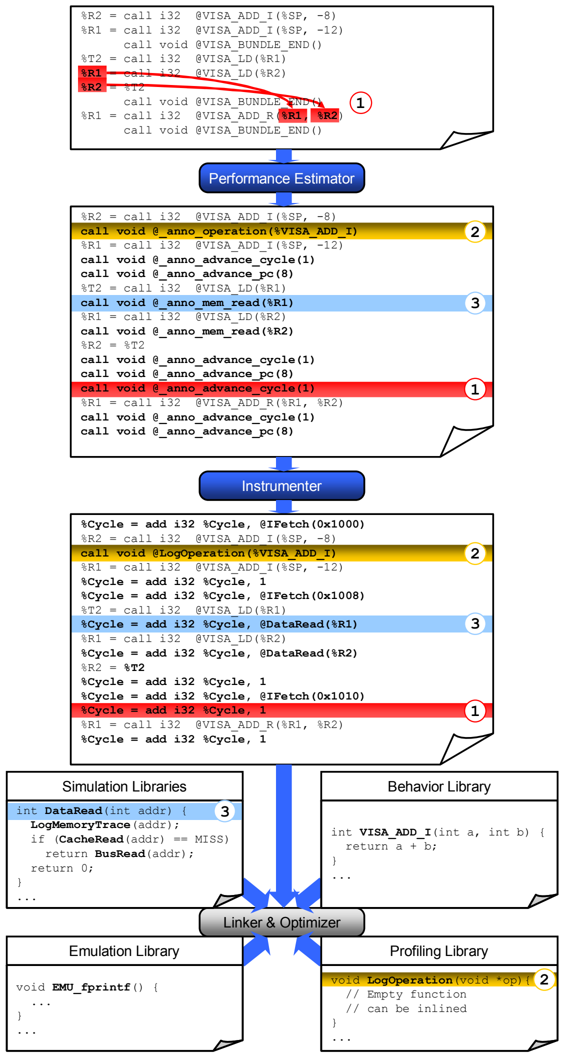
Last but not least, practically, not all custom instructions in an ASIP can be automatically utilized during compiler code generation. In this case, a retargetable assembly translator is provided so that the user can write "normal" assembly instead of having to program in virtual assembly.

3.2 Performance Estimation and Instrumentation

The performance of an application can be approximated using the following formula, in which $SCycle_n$ is the estimated static cycle cost of basic block n and $DCycle_m$ is the dynamic cycle cost when event m happens.

$$Cycle_{total} = \sum_{n=BB_0}^{BB_i} SCycle_n \times Count_n + \sum_{m=Event_0}^{Event_j} DCycle_m$$

To estimate the static cost of each basic block, optimistic static scheduling is performed during the *performance estimation* phase. Based on the architecture description, it identifies data hazards that can happen if a basic block is executed with no dynamic event (e.g., cache miss) taken into consideration. The analyzed static costs are annotated to the virtual assembly on a per-instruction basis, as shown in Figure 3, where the virtual assembly generated from Figure 2 undergoes further processing. In this example, two pipeline hazards (shown in ① of the figure) are detected in



```
%R2 = call i32  @VISA_ADD_I(%SP, -8)
%R1 = call i32  @VISA_ADD_I(%SP, -12)
      call void @VISA_BUNDLE_END()
%T2 = call i32  @VISA_LD(%R1)
%R1 = call i32  @VISA_LD(%R2)
%R2 = %T2
      call void @VISA_BUNDLE_END()
%R1 = call i32  @VISA_ADD_R(%R1, %R2)
      call void @VISA_BUNDLE_END()
```

```
%R2 = call i32  @VISA_ADD_I(%SP, -8)
call void @_anno_operation(%VISA_ADD_I)
%R1 = call i32  @VISA_ADD_I(%SP, -12)
call void @_anno_advance_cycle(1)
call void @_anno_advance_pc(8)
%T2 = call i32  @VISA_LD(%R1)
call void @_anno_mem_read(%R1)
%R1 = call i32  @VISA_LD(%R2)
call void @_anno_mem_read(%R2)
%R2 = %T2
call void @_anno_advance_cycle(1)
call void @_anno_advance_pc(8)
call void @_anno_advance_cycle(1)
%R1 = call i32  @VISA_ADD_R(%R1, %R2)
call void @_anno_advance_cycle(1)
call void @_anno_advance_pc(8)
```

```
%Cycle = add i32 %Cycle, @IFetch(0x1000)
%R2 = call i32  @VISA_ADD_I(%SP, -8)
call void @LogOperation(%VISA_ADD_I)
%R1 = call i32  @VISA_ADD_I(%SP, -12)
%Cycle = add i32 %Cycle, 1
%Cycle = add i32 %Cycle, @IFetch(0x1008)
%T2 = call i32  @VISA_LD(%R1)
%Cycle = add i32 %Cycle, @DataRead(%R1)
%R1 = call i32  @VISA_LD(%R2)
%Cycle = add i32 %Cycle, @DataRead(%R2)
%R2 = %T2
%Cycle = add i32 %Cycle, 1
%Cycle = add i32 %Cycle, @IFetch(0x1010)
%Cycle = add i32 %Cycle, 1
%R1 = call i32  @VISA_ADD_R(%R1, %R2)
%Cycle = add i32 %Cycle, 1
```

```
int DataRead(int addr) {
  LogMemoryTrace(addr);
  if (CacheRead(addr) == MISS)
    return BusRead(addr);
  return 0;
}
...
```

```
int VISA_ADD_I(int a, int b) {
  return a + b;
}
...
```

```
void EMU_fprintf() {
  ...
}
...
```

```
void LogOperation(void *op){
  // Empty function
  // can be inlined
}
...
```

Figure 3: Example of performance estimation and instrumentation

the virtual assembly, and one extra cycle has to be spent on solving these hazards. This extra cycle consumption is annotated to the virtual assembly. In turn, the annotation can be processed by a machine-independent instrumenter, which produces extra code to accumulate a global cycle counter (`Cycle`).

② of the figure shows an example of using annotation and instrumentation to support profiling. The type of each operation can be annotated, and then profiling code can be injected to log the execution count of each type. The profiling code is represented as function calls, which invokes profiling libraries developed in C and C++. Multiple choices of implementation can be selected by the user via linking with different libraries to meet various execution speed/profiling requirements. In general, the more complex the implementation, the slower the execution. The profiling functions can also be completely empty, so that the corresponding function calls can be eliminated thanks to the powerful LLVM interprocedural optimizations.

Dynamic events, another source of cycle consumption, can be simulated at runtime with the help of annotation. For example, the memory accesses are annotated (③), based on which extra code (`DataRead(%R1)`) is instrumented so that simulation libraries can be invoked. In this case, `DataRead` first logs a memory access to the profile, then calls another simulation library to perform cache simulation. If cache miss happens, a transaction is simulated using a bus simulator. Naturally, different implementations are also available for each of these functions. Moreover, through the bus simulation interface, multiple *TotalProf* instances (even mixed with ISSs) can be connected together.

Furthermore, API emulation libraries are also provided so that the system operations, e.g., file writing, can be supported.

3.3 Retargeting TotalProf

TotalProf can be retargeted using several different approaches. For profiling purposes, the *Application Binary Interface* (ABI) level compatibility does not have to be rigorously met. Consequently, retargeting *TotalProf* is extremely easy, compared to the traditional retargetable compilers.

3.3.1 Architecture Description

As shown in Figure 4, the architecture description in *TotalProf* consists of two parts: a *compiler machine description* that includes a *datapath description*, and an *instruction behavior description*.

In *TotalProf*, a virtual backend is implemented using the LLVM backend subsystem. The code selection, pre-RA scheduling and register allocation are implemented in the same way like most of the existing LLVM backends. A completely redesigned post-RA scheduler is developed in order to generate VLIW virtual assembly. The post-RA scheduler is driven by a *datapath description*, including inputs and outputs of each instruction, instruction latency tables and a reservation table. The datapath description is also used for the static performance estimation. Since the virtual assembly must be executed, the behavior of each instruction has to be provided, which can be described in C and C++. *TotalProf* can be fully-retargeted or semi-retargeted, as elaborated in the following.

3.3.2 Semi-Retargeting TotalProf

Although the architecture description contains more information than a compiler machine description, *TotalProf* can be easily retargeted through modifying some abstracted features, because no ABI-level compatibility has to be rigorously promised. This approach is also called *semi-retargeting* in this work.

A VLIW architecture description template constituting a large variety of instructions is the starting point of semi-retargeting. New custom instructions can be introduced and the existing instructions can be disabled. Calling conventions and the number of general purpose registers can also be easily changed. By modifying the instruction latency tables and the reservation table, the model of the processor's datapath can be easily configured. The tables can be directly modified or configured via adjusting the number of execution units for the sake of simplicity. As will be shown later, semi-retargeting is powerful enough to capture the performance characteristics of a large variety of target architectures, including ASIPs, DSPs and VLIW machines.

3.3.3 Fully-Retargeting TotalProf

Although good accuracy of semi-retargeting has been demonstrated, fully-retargeting *TotalProf* is also possible in favor of higher fidelity. In case that a LLVM backend for the target architecture already exists, one easy way to fully-retarget *TotalProf* is to reuse the majority of the machine description of the LLVM backend and to only change the syntax of each instruction so that virtual assembly instead of machine assembly is emitted. Afterwards, the instruction behavior and datapath descriptions can be provided to complete the retargeting.

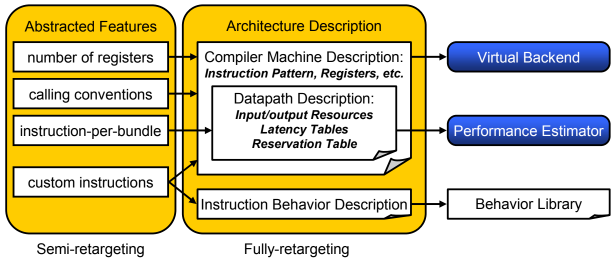


Figure 4: Retargeting *TotalProf*

4. RESULT AND CASE STUDY

This section first presents the experimental results of *TotalProf* to evaluate the execution speed, performance estimation accuracy, and retargetability. Then, two case studies on VLIW processor and MPSoC design space exploration are discussed. All the experiments are undertaken on a *Fedora Core 4 Linux* computer with Athlon 5200+ processor and 4 GB of memory. A number of embedded applications are used in the experiments.

4.1 Evaluation 1: Speed and Accuracy

The speed and accuracy of *TotalProf* is evaluated for the MIPS-4K architecture, because this architecture is also supported by the SLPE-based *micro-profiler*, so that a fair comparison can be made. A cycle-accurate MIPS ISS with built-in profiling functionality is used as a reference. This ISS is modeled with the *CoWare ProcessorDesigner* (based on [27]) and uses the *Just-In-Time Cache Compiled* (JIT-CC) simulation technique [26]. To use the ISS, the applications must be compiled. The MIPS-GCC (2.95.3) compiler is employed with O3 optimization threshold selected.

TotalProf is semi-retargeted to the MIPS architecture. The LLVM interprocedural optimizations are disabled so that the estimated result is closer to that of the MIPS-GCC. One important remark is that this evaluation does **not** intend to pretend that one compiler can be used to estimate a different one. Instead, the **only** purpose is to show that SLPE with a virtual backend is much better than without. Indeed, as a source code profiler, there is no generic approach to link with the compiler used in a product, but nevertheless, employing fast profiling as early as possible is still helpful in many scenarios, even if it is not exactly accurate.

4.1.1 Speed Comparison

To give an impression of the execution speed of *TotalProf*, the execution time of different approaches are measured. These results are represented in *Million Instructions Per Second* (MIPS) that is computed via dividing the number of *instructions* obtained from the CA ISS by the execution time of each approach. Note, that due to this calculation, sometimes the result may appear to extraordinarily large.

As shown in Figure 5, *GNU gprof* [11] is only slightly slower than the native execution on the host. Valgrind [25]

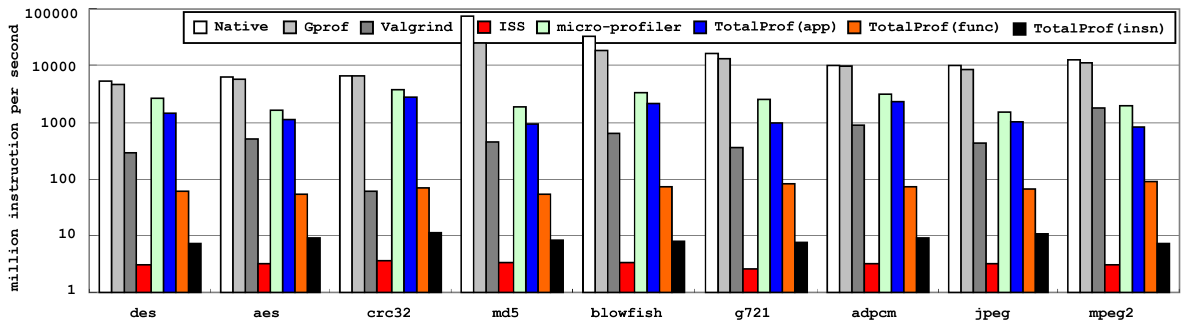


Figure 5: Speed Comparison

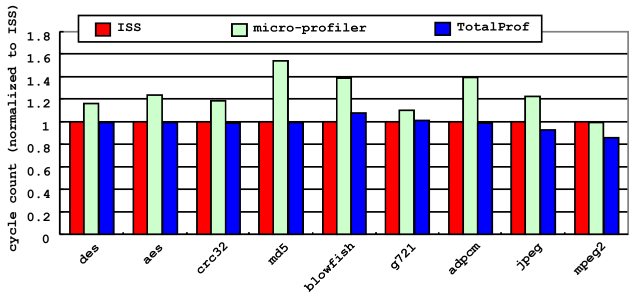
(with call graph generation but no cache simulation) slows the execution speed down significantly. All these three approaches do not provide target architecture specific performance information since they are not for cross profiling. However, the results are listed here as an additional information.

The execution speed of *TotalProf* heavily depends on the threshold of profiling, which can be controlled with many options. Three typical profiling levels are defined to enable the evaluation. They are (1) application-level SLPE, abbreviated as `app`, (2) function-level profiling (`func`), which records performance and memory access number for the invocation of each function, and (3) instruction-level profiling (`insn`), which not only performs all the profiling of the function-level one but also generates statistics such as execution count for each type of instruction. Cache simulation is performed in none of them. The execution speed of these three levels is compared with that of the JIT-CC ISS and the *micro-profiler* in Figure 5. At the `app`-level, *TotalProf* is only 1.67 times slower than the *micro-profiler* doing a similar profiling job. Meanwhile, *TotalProf*(`app`) is 456.6 times faster than the ISS. Note, that the accuracy of all these three levels are exactly the same, and the slowdown of the `func` and `insn`-levels is only caused by the profiling information generation. Nevertheless, the experiment shows at least the speed of `func`-level profiling can be improved in our future work, because of the observation that *GNU gprof*, which generates similar profiling information, is only marginally slower than the native execution.

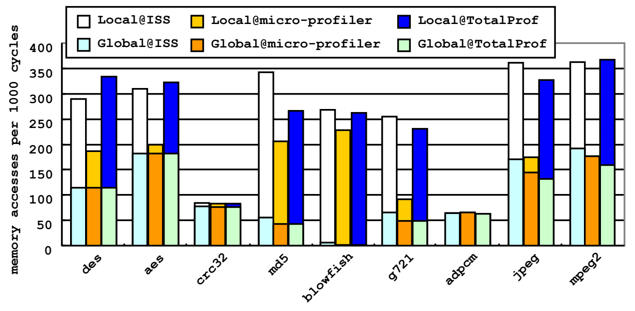
4.1.2 Accuracy Comparison

Figure 6 compares the generated profiles on application performance (normalized to that of the ISS), global/local memory accesses and instruction/data cache simulation results. On average (in this paper, standard deviation computed against reference values), *TotalProf* only introduces 5.8% error rate on the estimated performance, which is much better than the *micro-profiler* (29.1%).

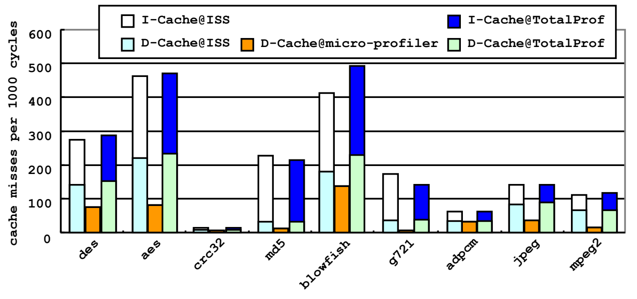
The global and local memory accesses per 1000 cycles of each application are given in Figure 6.(b), indicating both *TotalProf* and the *micro-profiler* are on a par in respect of global memory access estimation. Note, that for `blowfish`, the ISS suggests 8 global memory accesses but both *TotalProf* and *micro-profiler* estimate 1 access per 1000 cycles. Although the error rate is large, the absolute error is negligible, as only 7 accesses are miscalculated in each 1000 cycles. The average error rates of *TotalProf* and *micro-profiler* (neglecting this anomaly) are 15.1% and 13.1% respectively.



(a) Performance estimation



(b) Global and local memory accesses



(c) Instruction and data cache misses

Figure 6: Accuracy Comparison

TotalProf also gives a fairly accurate local memory access estimation (13.1%). However, the local memory accesses estimated with *micro-profiler* are distorted (67.9%) since it is not capable of simulating the effect of register allocation, due to the absence of compiler backend simulation.

To further evaluate the accuracy of memory profiles, cache simulation is applied as it is sensitive to memory access pat-

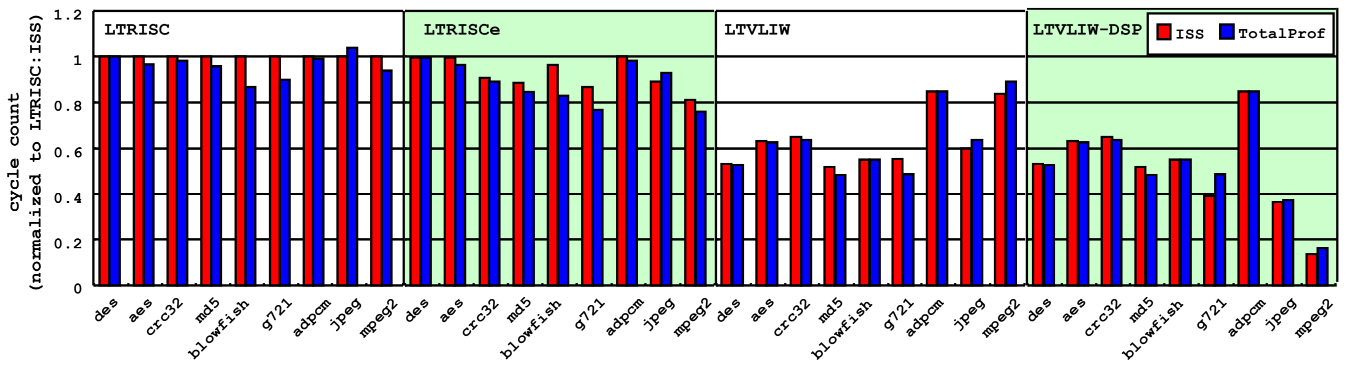


Figure 7: Retargetability evaluation results

terns. *TotalProf* shows a good accuracy on data cache simulation (9.5% error), but *micro-profiler* has an exacerbated 52.4% error rate due to the inaccurate local memory access estimation. Moreover, *TotalProf* can perform instruction cache simulation (10.4% error) since the instrumentation happens after linking of all compilation units, when the address of each virtual instruction can be assessed. The same feature cannot be supported by *micro-profiler* as well as many of other SLPE tools.

There is an interesting observation that *micro-profiler* more accurately reports the performance of the application `mpeg2`. However, the facts are as follows: (1) *Micro-profiler* tends to over-estimate the cost of instructions since its optimizations are not as strong as those of MIPS-GCC, as can be observed in Figure 6.(a). (2) *Micro-profiler* fails at estimating most of the local memory accesses for this application. (3) The above two factors are nullified in some sense, rendering an apparently "more" accurate estimation.

4.2 Evaluation 2: Retargetability

To further evaluate the retargetability, *TotalProf* is retargeted to four different embedded architectures, which are enumerated as follows.

1. **LTRISC** – a fixed-point RISC processor provided by *CoWare* as a template architecture for ASIP design. It has a 5-stage pipeline with no interlocking, therefore `nop`s have to be introduced to solve pipeline hazards.

2. **LTRISCe** – an enhanced version of LTRISC. Interlocking is used to solve pipeline hazards, and a branch predictor is introduced.

3. **LTVLIW** – a VLIW architecture with a RISC-like instruction set – another template provided by *CoWare* for ASIP design. It has a 5-stage pipeline with no interlocking, and 4 instructions can be issued per cycle.

4. **LTVLIW-DSP** – LTVLIW enhanced with DSP instructions, e.g., MAC instruction.

These processors cover a large variety of embedded architecture features, such as interlocking, branch prediction, VLIW architecture and DSP instruction. *TotalProf* is semi-retargeted to them, and Figure 7 shows the results of performance estimation in comparison with the instruction set simulation of the applications compiled with the corresponding LLVM compilers.

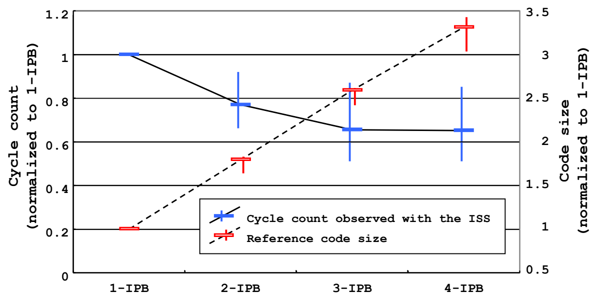
Since both the target compilers and the performance estimation are based on the same LLVM infrastructure, the results show an expected good accuracy of 6.1%, 5.9%, 3.2% and 3.3% for these four processor architectures separately. These results prove that the diversity of target processor architectures can be successfully captured with the virtual backend enhanced SLPE tool – *TotalProf*. At the same time, the speed advantage of SLPE is largely preserved, as the average execution speed of these *TotalProf*s is 1.35 GIPS, which is much higher than the JIT-CC technique powered ISSs at 5.3 MIPS averagely.

4.3 Case Study 1: VLIW Design Space Exploration

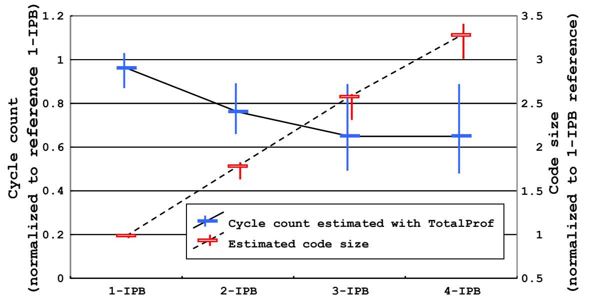
One advantage of using *TotalProf* instead of ISS for DSE is that the former can be easily constructed and modified. The reason is twofold: First, although ISSs can be automatically generated using ADLs (as will be discussed in section 5), describing and modifying an architecture is non-trivial. Second, most of the ADLs are not compiler-oriented, therefore cannot be easily used to explore compiler-oriented design variants.

This case study presents how *TotalProf* can facilitate a common VLIW design problem: deciding the number of *Execution Units* (EUs), which is a critical parameter of a VLIW processor. The more EUs, the higher instruction-level parallelism can be supported by the hardware. However, if the optimizing compiler cannot utilize the additional EUs by exploiting parallelism from the applications, the introduction of the additional EUs will only increase the cost and energy consumption of the VLIW processor. Moreover, if the VLIW processor cannot issue varying number of instructions, the introduction of the additional EUs can also have a negative impact on the code size.

The starting point of this case study is the default LTVLIW architecture provided by *CoWare* in their product *ProcessorDesigner*. This architecture is modeled in the LISA ADL [27]. Assembler/disassembler, linker, cycle-accurate ISS and RTL hardware description (e.g., VHDL) can be automatically generated using the *ProcessorDesigner*. By describing some compiler generation rules (already provided along with this model), a compiler can be generated. Using this tool, many important aspects, such as performance, number of memory accesses, code size, can be directly evaluated. The architecture description is modified to evaluate the performance and code size of the applications used in subsection 4.1 and 4.2. However, the modification is non-



(a) Reference performance and code size



(b) Estimated performance and code size

Figure 8: Results of VLIW design space exploration

trivial and error-prone. For example, changing the number of EUs from 4 to 2 involves 428 lines of code modification in 5 different files.

On the other hand, configuring *TotalProf* to estimate the performance of the VLIW processor with different numbers of EUs is extremely easy. Changing the number of EUs from 4 to 2 only involves 5 lines changes. Figure 8 shows the performance and code size of the applications obtained using the LISA model (with the generated cycle-accurate ISS) and the semi-retargeted *TotalProf*s. The average, maximum, and minimum values are plotted on the figures. At the same time, the execution speed of *TotalProf* is much higher than using the ISSs.

Although *TotalProf* is not a design tool to develop an architecture or its compiler/simulator, its use can help the architect by reducing the design space significantly, so that the effort of the further fine-grained DSE using the traditional approaches can be reduced. In this case study, LLVM compiler is also used to generate LTVLIW assembly, which is also an important reason why the results are so close. However, even if a different compiler is chosen for the architecture, such an early estimation is still helpful in reducing the design space that has to be explored later.

4.4 Case Study 2: MPSoC Design Space Exploration

MPSoCs have emerged in the embedded domain to solve the performance-power dilemma. However, application specific MPSoC design is still a difficult practice. This is due to the enormous complexity of both the multi-processor software and hardware. The lack of fast and accurate profiling tools has prevented software developers from fully utilizing the power of multi-processing, therefore has prohibited hardware developers from evaluating their design variants with decent workloads. The introduction of *TotalProf* attempts to alleviate this situation. This case study is performed in the following context: A H.264 baseline decoder application is parallelized by software developers into a scalable application, i.e., engaging more processors can improve the performance of the application. The software has been evaluated on multi-core hosts where the scalability has been demonstrated. However, there are two questions that need to be answered: First, is the software truly scalable on a properly configured embedded MPSoC platform? Second, what is the best hardware configuration to execute the software?

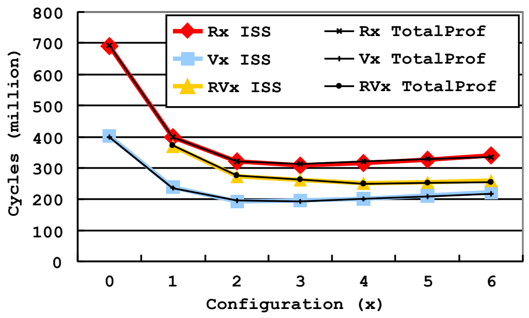


Figure 9: Results of MPSoC DSE

To answer these questions, MPSoCs with various numbers and types of *Processing Elements* (PEs) have to be evaluated. This requires a configurable MPSoC virtual simulation platform to be built. However, developing such a MPSoC virtual platform is extremely difficult and cannot be available early enough, rendering both the software team and the hardware team to possibly waste their time on deviated development directions.

In this case study, the details of the design variants are as follows: The MPSoC contains a number of RISC (LTRISC) and VLIW (LTVLIW) PEs connected with an AMBA bus. Maximally 7 PEs can be put on the chip. Each PE has its own local memory, while synchronization and communication is done using a shared memory. The H.264 application is parallelized into a *master* task (that undertakes control and some computation jobs) and a number of *slave* tasks for macroblock decoding that is deemed the most computation-intensive subtask. In each MPSoC design variant, a dedicated PE is used to execute the master task, and the slave tasks are mapped to the rest of the PEs, which are in turn called *slave PE*s. The design space consists of the following configurations:

- **RISC MPSoC**: The master task is executed on one RISC PE and the slave tasks are mapped to a number (`x`) of RISCs. Each design variant is named as `Rx`.
- **VLIW MPSoC**: Similarly, VLIW PEs are used for executing both the master and slave tasks, and each design variant is called `Vx` accordingly.
- **RISC-VLIW MPSoC**: A RISC is used to execute the master task and a number (`x`) of VLIW PEs are dedicated to executing the slave tasks. Each design variant is called `RVx`.

To explore this design space, *TotalProf* is semi-retargeted to the RISC and VLIW processors. Multiple instances of *TotalProf* are connected with *SystemC* bus/memory models to enable fast and accurate MPSoC profiling. Figure 9 shows the evaluation results. Compared to an ISS-based virtual platform that is available later, the accuracy of the *TotalProf*-based virtual platform is very good. The profiling indicates that the introduction of the first and second slave PEs can significantly improve the performance, but further performance improvement cannot be achieved with more slave PEs engaged. The profiling also unveils the underlying reason: Their are too many bus transactions and the chance of bus contention increases rapidly when more PEs are introduced.

The *TotalProf*-based virtual platform executes fast enough, and software developers can use it in their daily work to iteratively evaluate their modifications. For example, to evaluate the H.264 software on one of the MPSoC configurations, the ISS-based virtual platform takes 12 to 46 minutes (depending on the actual configuration), while the *TotalProf*-based only takes 11 to 42 seconds. In design space exploration, the architect needs to evaluate all the design variants. The ISS-based virtual platform takes about 8 hours in this case, while the *TotalProf*-based only consumes 7 minutes.

Nevertheless, this case study also shows a limitation of *TotalProf* that it only supports application profiling. If the MPSoC contains an operating system for task scheduling, the current *TotalProf* cannot handle it. It is our future work to improve *TotalProf* with task scheduling functionality.

5. FURTHER DISCUSSION

5.1 Comparison to Instruction Set Simulation

As reviewed in section 2, the instruction set simulation approach of highest execution speed is binary translation. Indeed, as indicated by Figure 1, *TotalProf* can also be regarded as a special binary translation based simulator that translates virtual assembly to a host executable. However, there are several differences. First, *TotalProf* is self-contained, as it does not rely on the availability of a target compiler. Second, *TotalProf* does not have to parse assembly files (or disassembled binary files) for various architectures, which is a major effort of retargeting a binary translator. Third, *TotalProf* benefits from the full power of the aggressive LLVM optimizing compiler, while the majority of binary translators cannot.

To elaborate why a traditional binary translator cannot benefit as much from optimizations as *TotalProf* does, it is essential to stress the difference between their translation processes. *TotalProf* generates virtual assembly, which is the IR of LLVM that can be directly optimized. Many works (e.g., [20]) also attempt to generate compiler IRs or HLLs (e.g., C language) to utilize existing optimizing compilers. However, many architectures have features that cannot be translated into high-level statements. For example, *indirect branch* instructions change the program counter to the values of the branch registers, and they are pervasively used in many architectures. However, in the majority of HLLs, only *indirect call*s are supported. Indirect branch instructions can be used to realize indirect calls, but can also be used for other purposes, such as implementing jump tables of `switch-case` statements. Given such problems, the generation of compiler IRs or HLLs is either infeasible or inefficient. As a consequence, many binary translators resort to using home-made IRs and optimizers, which are normally not as powerful as the counterparts of modern optimizing compilers.

5.2 Assembly Translation and Hybrid Simulation

As a source code profiler, *TotalProf* does not process target binaries, therefore cannot support the use of inline assembly or third-party libraries per se. Two methods are introduced to address this issue.

The first method is the aforementioned (subsection 3.1) retargetable assembly-to-virtual-assembly translator. Even if the C source code of a function is not available, virtual assembly can still be generated using the translator. Although not capable of translating indirect branch instructions, the translator can handle the majority of cases.

To support the other cases, hybrid simulation is developed. When a function that cannot be translated into virtual assembly is called, an instruction set simulator can be invoked to process the function. This concept can be regarded as a simplified (and less powerful) version of [10].

5.3 Comparison to Architecture Description Languages

Architecture Description Languages (ADLs) [23] are widely used in *Electronic System Level* (ESL) design. The majority [27, 28, 12] of the state-of-the-art ADLs dedicate to describing the structure of architectures, with the behavior of each instruction affiliated. As already mentioned in case study 1, a structural ADL can be used to model an architecture then both the software toolkit and the RTL description can be generated. However, if the architect wants to evaluate some design variants prior to development, modeling and modifying such kind of ADLs are not an easy task.

Trimaran [35] uses the HMDES language to describe an architecture. It is compiler-oriented, therefore except for the behavior description, the rest of the description is similar to a compiler machine description. Using *Trimaran*, compiler-oriented DSE can be more easily performed. The downside of compiler-oriented ADLs is that they are more difficult to use by hardware developers with little compiler background. In some sense, *TotalProf* is similar to *Trimaran*. However, *TotalProf* significantly alleviates the efforts of retargeting, and the execution speed of *TotalProf* is 3 orders of magnitude higher than that of *Trimaran*.

6. CONCLUSION

This paper introduces *TotalProf*, an application source code cross profiler that is devoted to high execution speed, good accuracy, and ease of retargeting. *TotalProf* introduces a novel virtual backend to the classic flow of source-level performance estimation to simulate the process of target compilation. An optimistic static scheduler is implemented and dynamic event simulation is encompassed. Experimental results show an average execution speed of more than one GIPS, while the estimation accuracy of both the processor datapath and the memory subsystem is close to instruction set simulation. The case studies on VLIW processor and MPSoC system design space exploration further highlight the application of *TotalProf*, indicating that retargeting *TotalProf* is much easier than modifying a model in an architecture description language. It is our future work to address the limitations of this approach, including developing a graphical user interface to retarget *TotalProf*, implementing automatic design space exploration, and supporting operating system profiling.

7. REFERENCES

[1] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous Profiling: Where Have All the Cycles Gone? *ACM Transaction on Computer Systems*, 15(4), 1997.

[2] D. Bartholomew. QEMU: a Multihost, Multitarget Emulator. *Linux Journal*, 2006(145):3, 2006.

[3] A. Bouchhima, P. Gerin, and F. Pétrot. Automatic Instrumentation of Embedded Software for High Level Hardware/Software Co-Simulation. In *ASP-DAC '09: Proceedings of the 2009 Conference on Asia and South Pacific Design Automation*, 2009.

[4] D. C. Burger and T. M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report CS-TR-1997-1342, 1997.

[5] B. D. Bus, B. D. Sutter, L. V. Put, D. Chanet, and K. D. Bosschere. Link-Time Optimization of ARM Binaries. In *LCTES '04: Proceedings of Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 211–220, 2004.

[6] L. Cai, A. Gerstlauer, and D. Gajski. Retargetable Profiling for Rapid, Early System-Level Design Space Exploration. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 281–286, 2004.

[7] J. Ceng, J. Castrillon, W. Sheng, H. Scharwächter, R. Leupers, G. Ascheid, H. Meyr, T. Isshiki, and H. Kunieda. MAPS: An Integrated Framework for MPSoC Application Parallelization. In *DAC '08*, pages 754–759, 2008.

[8] E. Cheung, H. Hsieh, and F. Balarin. Fast and Accurate Performance Simulation of Embedded Software for MPSoC. In *ASP-DAC '09*, 2009.

[9] L. Eeckhout, K. de Bosschere, and H. Neefs. Performance analysis through synthetic trace generation. In *ISPASS '00: Proceeding of IEEE International Symposium on Performance Analysis of Systems and Software*, 2000.

[10] L. Gao, S. Kraemer, R. Leupers, G. Ascheid, and H. Meyr. A Fast and Generic Hybrid Simulation Approach using C Virtual Machine. In *CASES '07: Proceedings of the 2007 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 3–12, 2007.

[11] S. L. Graham, P. B. Kessler, and M. K. McKusick. Gprof: A Call Graph Execution Profiler. *Proceeding of SIGPLAN Symposium on Compiler Construction*, 17(6):120–126, 1982.

[12] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. EXPRESSION: a Language for Architecture Exploration through Compiler/Simulator Retargetability. In *DATE '99: Proceedings of the Conference on Design, Automation and Test in Europe*, 1999.

[13] Y. Hwang, S. Abdi, and D. Gajski. Cycle-Approximate Retargetable Performance Estimation at the Transaction Level. In *DATE '08*, pages 3–8, 2008.

[14] Intel VTune. `software.intel.com/en-us/intel-vtune/`.

[15] D. Jones and N. Topham. High Speed CPU Simulation Using LTU Dynamic Binary Translation. In *HiPEAC '09: Proceeding of Conference on High Performance Embedded Architectures and Compilers*, 2009.

[16] K. Karuri, M. A. A. Faruque, S. Kraemer, R. Leupers, G. Ascheid, and H. Meyr. Fine-Grained Application Source Code Profiling for ASIP Design. In *DAC '05*, pages 329–334, 2005.

[17] D. Kim, J. Eom, and C. Park. L4oprof: a performance-monitoring-unit-based software-profiling framework for the l4 microkernel. *SIGOPS Operating System Review*, 41(4):69–76, 2007.

[18] M. Lajolo, M. Lazarescu, and A. Sangiovanni-Vincentelli. A Compilation-Based Software Estimation Scheme for Hardware/Software Co-Simulation. In *CODES '99: Proceedings of the seventh International Workshop on Hardware/Software Codesign*, pages 85–89, 1999.

[19] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO '04: Proceedings of the International Symposium on Code Generation and Optimization*, page 75, 2004.

[20] M. T. Lazarescu, J. R. Bammi, E. Harcourt, L. Lavagno, and M. Lajolo. Compilation-Based Software Performance Estimation for System Level Design. In *HLDVT '00: Proceedings of the IEEE International High-Level Validation and Test Workshop*, page 167, 2000.

[21] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 190–200, 2005.

[22] T. Meyerowitz, A. Sangiovanni-Vincentelli, M. Sauermann, and D. Langen. Source-Level Timing Annotation and Simulation for A Heterogeneous Multiprocessor. In *DATE '08*, pages 276–279, 2008.

[23] P. Mishra and N. Dutt. *Processor Description Languages, ISBN: 1875-9661*. Morgan Kaufmann Publishers Inc., 2008.

[24] T. Moseley, A. Shye, V. J. Reddi, D. Grunwald, and R. Peri. Shadow Profiling: Hiding Instrumentation Costs with Parallelism. In *CGO '07*, 2007.

[25] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *PLDI '07*, pages 89–100, 2007.

[26] A. Nohl, G. Braun, O. Schliebusch, R. Leupers, H. Meyr, and A. Hoffmann. A Universal Technique for Fast and Flexible Instruction-Set Architecture Simulation. In *DAC '02*, 2002.

[27] S. Pees, A. Hoffmann, V. Zivojnovic, and H. Meyr. LISA – Machine Description Language for Cycle-Accurate Models of Programmable DSP Architectures . In *DAC '99*, 1999.

[28] J. V. Praet, D. Lanneer, W. Geurts, and G. Goossens. nML: A Structural Processor Modeling Language for Retargetable Compilation and ASIP Design. *Processor Description Languages*, pages 65–94, 2008.

[29] W. Qin, J. D'Errico, and X. Zhu. A Multiprocessing Approach to Accelerate Retargetable and Portable Dynamic-compiled Instruction-set Simulation. In *CODES+ISSS '06: Proceeding of Conference on Hardware/Software Codesign and System Synthesis*, 2006.

[30] M. Reshadi, P. Mishra, and N. Dutt. Instruction Set Compiled Simulation: A Technique for Fast and Flexible Instruction Set Simulation. In *DAC '03*, 2003.

[31] A. Sahu, M. Balakrishnan, and P. R. Panda. A Generic Platform for Estimation of Multi-threaded Program Performance on Heterogeneous Multiprocessors. In *DATE '09*, 2009.

[32] J. Schnerr, O. Bringmann, A. Viehl, and W. Rosenstiel. High-Performance Timing Simulation of Embedded Software. In *DAC '08*, pages 290–295, 2008.

[33] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder. Discovering and Exploiting Program Phases. *IEEE Micro*, 2003.

[34] A. Srivastava and A. Eustace. ATOM: A System for Building Customized Program Analysis Tools. In *PLDI '94*, pages 196–205, 1994.

[35] Trimaran. `www.trimaran.org`.

[36] K. Vaswani, M. J. Thazhuthaveetil, and Y. N. Srikant. A Programmable Hardware Path Profiler. In *CGO '05*, pages 217–228, 2005.

[37] M. T. Yourst. PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. In *ISPASS '07*, 2007.

[38] J. Zhu and D. D. Gajski. A Retargetable, Ultra-Fast Instruction Set Simulator. In *DATE '99*, 1999.