Fine-grained Application Source Code Profiling for ASIP Design

Kingshuk Karuri, Mohammad Abdullah Al Faruque, Stefan Kraemer,
Rainer Leupers, Gerd Ascheid, Heinrich Meyr

Institute for Integrated Signal Processing Systems
RWTH Aachen, Germany

ABSTRACT

Current Application Specific Instruction set Processor (ASIP) design methodologies are mostly based on iterative architecture exploration that uses Architecture Description Languages (ADLs) and retargetable software development tools. However, for improved design efficiency, additional pre-architecture exploration tools are required to help narrow-down the huge design space and making coarse-grained Instruction Set Architecture (ISA) decisions before detailed ADL modeling. Extensive application code profiling is the key in such early design stages. Based on a novel code instrumentation technology, we present a *micro-profiling* approach that fills the current gap between source-level and instruction-level profilers and combines their advantages w.r.t. speed and accuracy. We show how the micro-profiler is embedded into an advanced ASIP design flow and justify its use in a case study to design an MP3 decoder ASIP.

Categories and Subject Descriptors: C.1.3 [Processor Architectures]: Other Architecture Styles

General Terms: Design, Performance

Keywords: Customizable Processors, ASIPs, Profiling, Codesign

1. INTRODUCTION

The need for high efficiency, combined with time-to-market pressure, are among the most important challenges in embedded system design today. Therefore, programmable and customizable ASIPs have become an attractive implementation technology [1, 2]. They offer efficiency and short design cycles at the same time. In contrast to off-the-shelf processor cores, ASIPs show dedicated functional units and machine instructions that speed up execution of the "hot spots" in a given application. For instance, a processor for DSP applications generally shows MAC units, dual memory banks, and address generation hardware, while a Network Processor provides instructions for fast data packet manipulation, routing table lookup etc. Simultaneously, the flexibility of programmable processors facilitates IP reuse and reduces the design risk. Recent results indicate that by extending a processor only with a few application-specific custom instructions one can accelerate code execution enormously at a very low area overhead [3, 4, 5].

Consequently, the amount of ASIP-related products in the IP and EDA industries has grown significantly in the last years. Companies like Tensilica (Xtensa), ARC (Tangent), CoWare (LISATek), and Target (Chess/Checkers) offer extensible cores and/or design tools for tuning processors towards given applications, and also IP vendors like ARM (OptimoDE) and MIPS (CorExtend) have extended their portfolio towards customizable processors. From the research perspective, significant focus has recently been on tools for automatic synthesis or customization of ASIP instruction set architectures (ISAs) (e.g. [6, 7]), and first commercial offerings have been announced in that area, too, e.g. Tensilica's XPRES technology as well as Stretch's software-configurable S5000 processor.

In accordance with Amdahl's Law, in order to make the frequent case fast, it is key in ASIP design to identify the hot spots in the application code, and to remove bottlenecks in the processor architecture correspondingly, e.g. by providing dedicated functional units and machine instructions. This requires extensive *application code profiling*. Profiling can take place at the source code (C/C++) level or at the assembly level. Though C-level profiling is a well accepted methodology, it still leaves a large gap to the actual implementation in the form of instructions executed on an architecture, which means low profiling accuracy. On the other hand, instruction-level profiling requires that at least a *model* (or virtual prototype) of the architecture, including an instruction set simulator, is already at hand during the time of architecture exploration. However, developing the detailed virtual prototype is already a time-consuming task. Given that ASIP design frequently starts with a plain C/C++ specification of the application and potentially a partially predefined or legacy IP core, it is desirable to provide "pre-architecture" exploration capabilities in order to narrow the huge design space, so as to increase design productivity.

The contribution of this paper is a novel fine-grained profiling technology, intended to *bridge the gap between traditional C-level and assembly-level profiling*, while combining advantages of both w.r.t. flexibility, speed, and accuracy. The proposed *micro-profiler* is conceived as a part of a workbench that guides the ASIP designer in early stages before committing to a detailed architecture. In addition, collected profiling statistics can be exported to instruction synthesis tools for further processing.

After a discussion of related work in section 2, the micro-profiling approach and its integration into the design flow are described in more detail in section 3, while section 4 outlines the implementation and section 5 presents the ways to export, view and use collected profiling information. In section 6, experimental results and application case studies are given, while section 7 concludes and mentions current limitations and future work.

2. RELATED WORK

The most widespread type of profiling is *source code level*

DAC 2005, June 13–17, 2005, Anaheim, California, USA.

profiling, mostly for C/C++ code. Standard tools like GNU gprof/gcov can be used in connection with C compilers to generate statistics about CPU time spent in different functions as well as execution frequencies of code lines. Such tools are traditionally used to *optimize the application code itself* by rewriting the hot spots for more efficient execution on the target machine. More recently, source-level profiling is also being adopted in ASIP design flows, in order to identify code fragments that benefit from hardware support via custom instructions. For the latter purpose, however, source-level profiling is not the ideal choice, due to the high abstraction level of C code:

1. Depending on the *programming style*, a single line of C code may consist of a number of different operations that map to many instructions at the machine code level. Hence, the granularity of C-level profiling is too coarse for ASIP ISA design.
2. In high-level languages like C/C++, *many operations are implicit* and not visible to the source-level profiler. This concerns e.g. implicit address computations and memory accesses, pointer scaling, and type casts. All of these are relevant for ISA design but are not captured at the C level.
3. A typical C compiler performs numerous *optimizing transformations* on the code. Statements may be rearranged for better performance, and operations may be substituted or even completely eliminated. These effects are neglected during C-level profiling, possibly leading to erroneous ISA design decisions.

Profiling can also be performed at the *assembly code level*. Such profilers are mostly machine-specific (e.g. SpixTool [8] for SPARC or VTune [9] for Intel architectures, cf. [10] for an overview). Other tools, such as LISATek [11] include retargetable assembly-level profilers, based on an ADL model of the target machine. While this type of profiling is definitely fine-grained enough for ISA design, it requires a detailed architecture model, from which an ISA simulator with an embedded profiler can be generated. Moreover, despite significant progress in instruction set simulation technology [12, 13], the speed gap of simulation vs. native compiled C code execution is still several orders of magnitude. Hence, assembly-level profiling is by far not as efficient as its source-level counterpart.

Our approach aims at combining the advantages of both classical profiling technologies, i.e., high speed and accuracy. The key idea is to apply source-level profiling, yet precisely counting *all* primitive operations during execution of an application. As outlined later, further dynamic execution statistics relevant for ASIP ISA design can be determined, too. Two recent works come close to this concept. The SIT toolkit [14] performs fine grained C-level profiling by exploiting C++ operator overloading capabilities. However, different C operators with similar instruction-level semantics are still counted separately. For instance, this concerns pointer indirection ("*ptr"), as well as array ("[]") and structure access ("->"), even though all of these eventually map to "LOAD" instructions. Moreover, logical operations ("&&, ||") are not lowered to the instruction level, causing potentially misleading profiling results. Similar restrictions apply to the retargetable SpecC profiler presented in [15]. Though that profiler has a larger scope than ours, e.g. explicitly covering component communication costs, its granularity w.r.t. counting primitive, assembly-like operations is relatively coarse.

3. MICRO-PROFILING APPROACH

The proposed *micro-profiler* (μP) is part of an advanced ASIP design flow that builds on state-of-the-art ADL based architecture exploration tools like [11, 16] (fig. 1). We assume the application C source code is given, and an ASIP

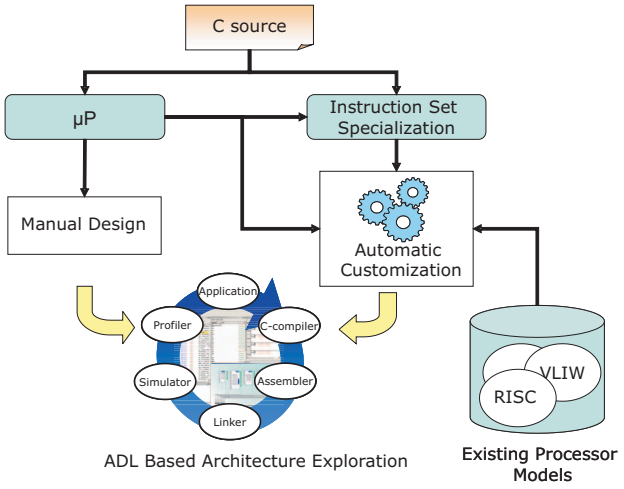

Figure 1: μP in an enhanced ADL based ASIP design flow

needs to be designed or customized for the application. In the pre-architecture stage, the designer needs to determine key dynamic execution statistics for early decisions on the ISA and the memory subsystem. This includes e.g. operation execution frequencies, cache hit rates, frequently used C data types together with their dynamic min/max values, as well as bit width of arithmetic operands and constants. Obviously, these data are extremely helpful in selecting the right accelerator functional units or designing an initial ISA. The μP determines these data by executing the C application code on the host, after automatic instrumentation as described in section 4.

The collected statistics may be used in two ways. A GUI front end of the μP presents the data to the designer in tabular and graphical form. In this scenario, the μP acts as a stand-alone workbench tool, guiding the designer during development of an initial ADL processor model. A set of *retargetable software tools* are then automatically generated from the ADL model for fine-grained (micro-architecture level) design space exploration. The final hardware implementation is developed (or automatically generated from the ADL model) if all the design constraints are met. In another (more automated) usage scenario, the profiling data are passed to an ISA synthesis tool. This tool, the detailed description of which is beyond the scope of this paper, applies an optimization algorithm similar to [6] in order to synthesize a limited number of complex custom machine instructions for highest speedup, based on a *weighted operation execution profile*. It generates ADL code for custom instructions that can be linked to existing ADL processor templates, e.g. a RISC core. This offers a direct path to micro-architecture level exploration and implementation with existing tools.

4. μP IMPLEMENTATION

Like most profilers, the μP is based on *code instrumentation*, i.e. it inserts additional code into the original C code. This code does not modify the program semantics, but only maintains internal data structures and counters for collecting profiling data during execution of the compiled and instrumented application. In order to overcome the problems of traditional C-level profilers mentioned in section 2, we apply different techniques from compiler construction:

1. The original C code is lowered to *Three Address Code*. Each line of such code contains at most one operation, which enhances the profiling granularity. We use the LANCE compiler [17] to generate an executable three address *Intermediate Representation (IR)* in C syntax. In order to minimize the execution time overhead

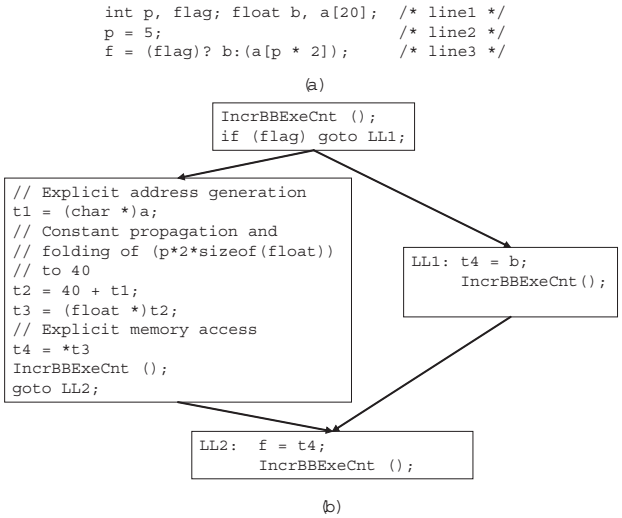

Figure 2: (a) Example C code and (b) Corresponding instrumented three address IR

of instrumented vs. original C code, the μP inserts *instrumentation code* whenever possible only at the end of each basic block. This code maintains *aggregate counters* for the entire block, while still being able to reconstruct statement-true information.

2. In the three address code, all primitive C-level operations, including type casts, pointer scaling etc. *are made explicit* and hence can be profiled like regular operations. All high-level operations are appropriately *lowered to a canonical form*, e.g. all memory accesses, including global variables, arrays, and structs, are mapped to explicit LOAD/STORE operations via pointers.

3. By exploiting the *built-in standard code optimizations* (e.g. constant propagation, dead code elimination) of LANCE that operate on the three address code, the μP can already emulate many code transformations likely to be performed later in the target specific C compiler. This increases the profiling accuracy.

Figure 2 shows a piece of C code and the corresponding three address IR to highlight the limitations of profiling at C source code level. The example code, in figure 2.(a), has embedded control flow due to the `?:` operator, hidden computations in the address generation of `a[p * 2]` and a *load* operation due to the array access. A source level profiler will not be able to profile these in detail. Since the execution frequencies of lines 2 and 3 are always same, a source level profiler will report them to be equally contributing to the application's run time, whereas in reality, line 3 is costlier.

Profiling at IR level solves these problems by the lowering operation. As can be seen from figure 2.(b), the control flow, address computation and memory access are explicit in the IR. It also prevents any false prediction by running high-level optimizations such as the propagation of the constant definition of `p` in line 2 in figure 2.(a) to the next line and then folding the expression `p * 2` to a constant. These optimizations ensure that the corresponding multiplication is not counted.

In the μP work-flow, a C application is run through the LANCE front end and code instrumenter, successively, to generate the *instrumented code*. The instrumented code contains IR code intermingled with extra *function calls* (such as the `IncrBBExeCnt` in figure 2) to collect profiling information. The definitions of these functions are available in a *profiler library* that performs the task of book-keeping during application execution. Since the instrumented code is a

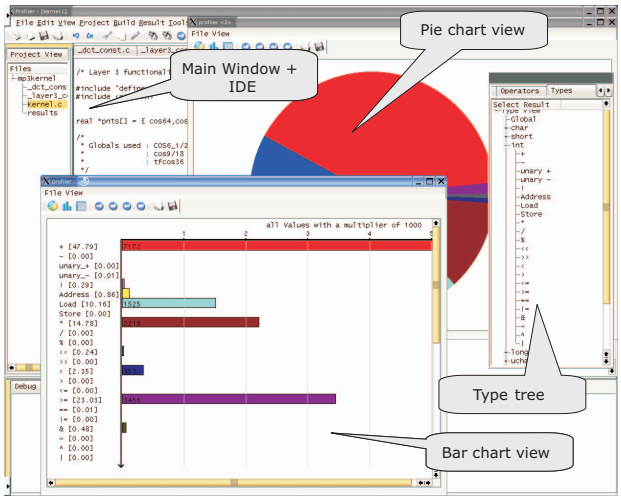

Figure 3: A GUI snapshot

subset of ANSI C, it can be easily compiled using any standard C compiler such as gcc making the whole system fully portable to any host system. The compiled code can then be linked with the profiler library and executed on the host machine to obtain profiling data.

5. EXPORTING AND USING PROFILING INFORMATION

5.1 Profiling options

The μP, as a stand-alone tool, produces a large amount of data which is then parsed and presented in convenient, user-readable forms such as tables, pie and bar-charts in a *Graphical User Interface (GUI)*. A snapshot of the GUI is presented in figure 3.

The μP can be run with a variety of profiling options that can be configured through the GUI. These options are listed below :

1. **Operator execution frequencies**: Execution count for each C operator per C data type in different functions and globally. This can help designers to decide which functional units should be included in the final architecture.

2. **Occurrences, execution frequencies and bit widths of immediates**: This allows designers to decide the ideal bit width for integral immediate values and optimal instruction word length.

3. **Conditional branch execution frequencies and average jump lengths**: This allows designers to take decisions in finalizing branch support hardware (such as branch prediction schemes, length of jump addresses in instruction words, zero-overhead loops etc.).

4. **Dynamic value ranges of integral C data types**: Helps designers to take decisions on data bus, register file and functional unit bit widths. For example, if the dynamic value range of integers is between 17 and 3031, then it is more efficient to have 16-bit, rather than 32-bit register files and functional units.

Apart from the above functionalities, μP can also be used to obtain cycle count estimates and execution frequency information for C source lines to detect application hot spots.

5.2 Profiling for optimal data-cache hierarchy design

The continually increasing performance gap between processors and memory systems has made memory hierarchy

configuration too important a task to be left alone for later stages of architecture design. Architectural performance is heavily dependent on cache behavior and *average memory access latency* for an application. Like many other dynamic execution characteristics of an application, the processor-memory traffic can also be monitored at the C source code level using μP. This information can be used in deciding the optimal *cache hierarchy configuration* at an early design phase by using *cache simulation* techniques.

The μP based cache simulation framework uses the well-known *trace-driven cache simulation* technique [18]. A C application is instrumented in such a way that it generates a *memory trace* containing *addresses* and *types* (i.e. read/write) of all data memory accesses during execution. The generated trace is simulated using the freely available trace-driven *dineroIV* [19] cache simulator. The framework allows the user to experiment with different cache-hierarchies and cache parameters (such as associativity, block size, cache size etc.) easily and quickly. The collected cache miss statistics can be used to decide an optimal memory system for the application in consideration.

In a real processor, data memory accesses can arise from five sources :

1. accessing local scalar variables/function parameters placed on function stack
2. register spilling
3. stack build-up and clean-up in function *prologue/ epilogue*
4. global/static scalar variables access
5. local/global *array element/structure field* access or *pointer dereferencing*.

The majority of the memory accesses result from the last two cases. Since LANCE lowers all global/static accesses, array element and structure field accesses to *pointer dereferences*, the number of memory accesses and the corresponding memory addresses can be easily profiled by adding instrumentation code before any *pointer dereference* operation in the three address IR. For usual RISC machines with large general purpose register banks, register spills are extremely rare and normally most of local scalar variables/function arguments can be fit into registers. Therefore, their contribution to the total data memory traffic is minimal. Since the data-cache behavior for any application depends on the memory access patterns (and not the actual memory addresses), memory traces obtained on a general purpose host machine mimics the cache behavior of an ASIP fairly accurately (as will be seen in subsection 6.2).

6. RESULTS AND APPLICATIONS

In order to evaluate the usefulness of the μP for ASIP design, we have done a number of experiments. The first two subsections focus on the speed and accuracy of μP w.r.t. *Instruction Set Simulation (ISS)* which is a prerequisite for it being used in early design space exploration. The last subsection shows how the tool can be used to customize an ASIP for a particular application.

6.1 Profiling speed

For fast design space exploration, the μP instrumented code needs to be at least as fast as the ISS of any arbitrary architecture. Preferably, it should be as fast as code generated by the underlying host compiler, such as *gcc*. Figure 4 (not drawn to scale) compares *average* speeds of instrumented code vs. *gcc (version 2.95.3)* generated code, and a fast compiled MIPS instruction accurate ISS (generated using LISATek tool suite, [11]) for the different configurations of μP.

As can be seen, the speed goals are achieved. The basic profiling options slow down instrumented code execution

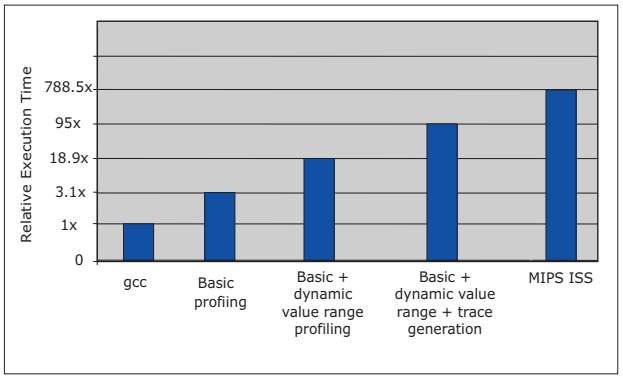

Figure 4: Relative speeds of gcc and μP generated code and MIPS instruction set simulation

vs. gcc by a factor of only 3. More advanced profiling options increase execution time significantly. However, even in the worst case, the instrumented code is almost an order of magnitude faster than ISS.

6.2 Profiling accuracy

Figure 5 shows the accuracy of μP based memory simulation for an ADPCM speech codec w.r.t. the LISATek on-the-fly [11] memory simulator (integrated into the MIPS ISS). The memory hierarchy in consideration has only one cache level with associativity 1 and block size of 4 bytes. The miss rates for different cache sizes have been plotted for both memory simulation strategies. As can be seen from the comparison, μP can almost accurately predict the miss rate for different cache sizes. This remains true as long as there is no or little overhead due to standard C library function calls. Since μP does not instrument library functions, the memory accesses inside binary functions remain un-profiled. This limitation can also be overcome if the standard library source code is compiled using μP.

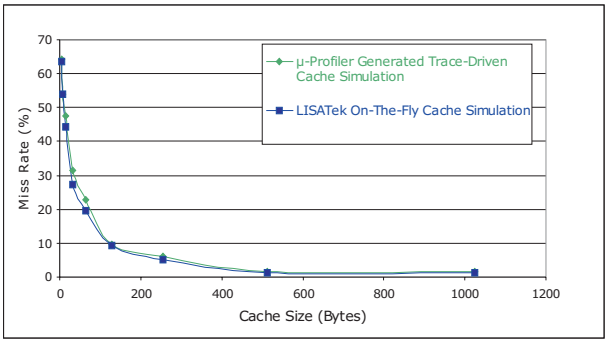

Figure 5: Miss-rate comparison between LISATek-on-the-fly and μP based cache simulation

The accuracy of μP in correctly estimating the cycle counts and execution frequencies of different operators for arbitrary target architectures is also of great importance. The cycle counts estimated by μP can be used to identify hot spots in an application or aid the instruction set specialization process. This is useful when the user is customizing an ASIP rather than designing one from scratch.

μP can be configured to obtain cycle count data of a partially predefined customizable architecture by using a rudimentary *retargeting* technique where a *user configurable cycle count weight* is assigned to each combination of C operators and types. Moreover, the user can also expand a C $< operator, type >$ pair to a list of other $< operator, type >$ pairs. For example, a negation operation can be mapped to a logical complement followed by an addition if the target architecture performs negation in this manner. Our tech-

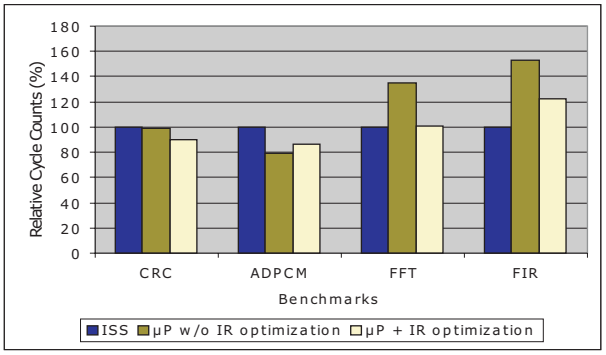

Figure 6: Comparison of cycle counts reported by ISS and μP with and without high level IR optimizations

nique is similar to the system-level performance estimation strategies presented in [22, 23] which are quite elaborate and more accurate, but are not conceived to guide the ASIP design process through profiling.

For an application consisting of n IR statements the cycle count estimate is given by the formula[1]:

$$Cycles = \Sigma_{i=1}^{n} E(S_i) \times W(S_i)$$

where $E(S_i)$ and $W(S_i)$ are the execution count and weight, respectively, for statement S_i. Figure 6 shows the relative cycle counts reported by cycle accurate ISS of a small RISC machine and μP (with and without running high level IR optimizations) normalized to ISS reported cycle count values. The average deviation from ISS without high level optimizations is 27%. But when high level LANCE optimizations are run on the IR code, the average deviation becomes much smaller (11%). This indicates that the μP can be fairly accurate in reporting cycle counts if it is properly configured to mimic the behavior of an *optimizing C compiler* for arbitrary architectures. It also highlights where source level profilers can go wrong in predicting application hot spots.

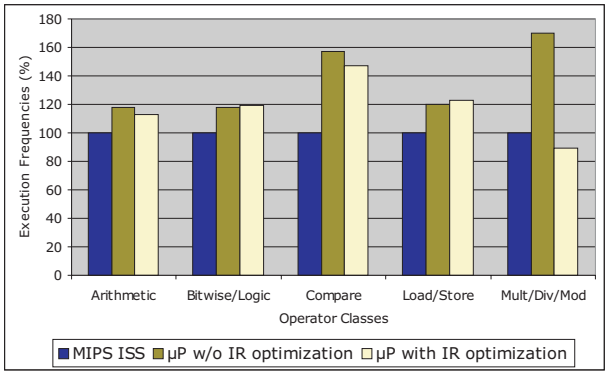

Figure 7: Comparison of actual and μP estimated operator execution frequencies with and without high level IR optimizations

Figure 7 shows the operator count comparisons obtained from MIPS instruction accurate simulation and instrumented code for the ADPCM benchmark. For the sake of convenience and brevity, we have subdivided the C operators into 5 categories. As can be readily seen, the average deviation from ISS is reasonable in this case, too. Moreover, the average deviation is lower (23%) with IR optimizations than without (36%).

[1]This applies mainly to RISC like single instruction issue processors. For multiple instruction issue processors, the code instrumenter has to statically analyze the potential amount of parallelism for higher accuracy.

6.3 Case study

The case study for μP has been to implement a small ASIP for well known *mpeg-layer3 (MP3)* audio decoding. We extracted the *frame decoding* kernel of the code (around 800 C source lines) from a publicly available implementation [20] of the standard. Around 80% of execution time, in any average run, is spent inside this kernel code.

We decided to customize a simple processor[2] for this application kernel using inputs from μP. We used automatically generated (by LISATek generators) software tools (C compiler, assembler, linker, loader, instruction set simulator) and hardware model for easy modification and comparison of results for different processor configurations. The original processor has 32-bit instruction words, 32-bit integer data type and functional units.

We started the exploration process by running the kernel through μP for several randomly selected frames, and analyzed the profiling data. Depending on the profiling information we introduced changes in the architectural model. After each incremental change, we obtained new cycle count and code size figures by retargeting the software tools, and obtained area and clock period information by synthesizing the generated hardware model using gate level synthesis tools (with a 0.18μ library).

	integral (%)	float (%)	pointer (%)	Total (%)
arithmetic	45	23	100	41
logical/bitwise	1	0	0	1
multiplication	14	18	0	12
comparison	22	2	0	2
load/store	18	57	0	34

Table 1: Average operator execution frequencies for frame decoding obtained via μP

As the first design step, we decided to analyze the average execution frequencies of different C operators in the kernel. The collected data are summarized in table 1. As can be readily seen, there is a considerable number of *single precision floating point* operations in the kernel. Our selected architecture, without an FPU, needed to emulate each floating point operation in software. Experiments with a floating point emulation library [21] indicated that such emulation could be at least two orders of magnitude slower than hardware implementation. μP generated cycle count figures suggested that, with a 100 times slower floating point emulator, around 60M cycles would be required to decode one single frame at 192 kbps bit rate. Therefore, to play 38 frames per second (as required by the MP3 standard) the processor would need 2280 M cycles per second, resulting in a very high clock frequency. This instantly suggested that a single precision FPU must be added to the architecture to meet the real-time constraints of the application.

Since inclusion of an FPU is costly in terms of area, we decided to defer this step and look for some other area saving optimizations. Analysis of the immediate value ranges, dynamic value ranges, branch profiling information and operator execution frequencies revealed some more useful data summarized below:

1. The integer comparisons are almost entirely due to the `>=` operator. This data immediately suggested elimination of all comparison operations except `>=` from the original ISA.
2. Bulk (more than 98%) of the immediate integral values, used in different operations, needed less or equal

[2]This simple processor is a basic RISC written in LISA 2.0 language [11]. It is called LTRISC and comes with the standard LISATek distribution.

to 8 bits for representation. Therefore, we decided to have only 8-bit immediates in the instruction set (rather than 12 and 16-bit wide immediates as was in the original architecture).

3. The average jump length only needed 8 bits for representation. So we decided to shorten the immediate jump length to 16 bits from 20 bits of the original architecture. This estimate is still fairly conservative.

4. The values of integral types was within the range between -7012 and 17664. This indicated that only 16-bit integral values should suffice in all cases.

Taking clues from the above information, we tried a new configuration with reduced jump length, shortened immediate value and fewer comparison operations in the architecture. With fewer instructions and shorter immediates and jumps, it was possible to re-arrange the instruction coding and reduce the instruction word length to 24 bits (instead of the original 32 bits). This immediately led to a saving of around 20% in code size. We also had some area reduction in instruction decoding logic and ALU, but it was minimal. Moreover, the cycle count increase for leaving out the comparison instructions was around 18%. So, this configuration did not seem promising at all.

As an experimental next step, we migrated from 32-bit to 16-bit ALUs and register files for integral operations following the dynamic value range information. This brought down the total area of the processor drastically to *8.82* K gates from *18.81* K gates of the original processor. Since floating point emulation is difficult to implement with 16-bit integers, we decided not to retarget the software tools for this configuration to obtain cycle count figures. Instead, we decided to use the area savings for implementing a 32-bit FPU with multiplier, adder, subtracter, comparator and eight 32-bit floating point registers. With the FPU, the average cycle count figures per frame came down to 410 K cycles. At this rate the architecture can decode 38 frames in 16.31 M cycles i.e. it needs a 16.31 MHz clock. To be on a safe footing (specially to run files with bit rates higher than 192 kbps), we decided to have a clock of 25 MHz (40 ns). Using this as a clock period constraint, we ran gate level synthesis tools to obtain the area and cycle length results for each intermediate processor configuration.

The final comparison of code size, average cycles per frame, cycle length and area is summarized in table 2. As can be readily seen, the final processor's clock is 16.08 ns slower than the original, but still meets the cycle length requirement of 40 ns. Additionally, in terms of cycles per frame, the final configuration is around 300 times faster than the original due to the newly added FPU. It also requires less amount of area (83% of original) and code size (14% of original). The code size savings are due to two reasons : migration from 32-bit to 24-bit instruction word and elimination of the floating point emulation routines from the code.

	K cycles/ frame	clock (ns)	area (K gates)	code size (KB)
original	132,649	23.60	18.81	83.08
w/o compare + 24-bit instr.	156,273	22.07	17.29	66.75
16 bit integer	NA	15.29	8.82	NA
with FPU	410	39.68	15.61	11.52

Table 2: Comparison of code size, area, clock length and cycle counts with various processor configurations

7. CONCLUSIONS AND FUTURE WORK

In this paper, we have introduced a novel profiling technology for designing application specific processors. The profiler can be used to accurately analyze the bottlenecks in designing or customizing ASIPs at the pre-architecture exploration stage. The application of this profiling technology in designing processors for real life applications has also been demonstrated using a small case study.

Currently, the hints generated from μP need to be manually analyzed and implemented for a processor. In the future, the μP can be coupled with a suitable back end so that the findings from profiling can be fully or semi-automatically translated to a high level processor architecture specification. Another unexplored area is the proper cost modeling for accurate estimation of the impact of μP generated hints without committing the decisions to an ADL model. This might significantly speed-up the overall design process through more effective pre-architecture exploration.

The types of profiling information currently collected are by no way complete. A number of advanced profiling options such as estimates of maximum heap and stack size and accesses to global and local variables can be added in future. This can help designers take more advanced architectural decisions such as the address generation requirements of a program. Additional case studies with non-RISC and multiple instruction issue architectures are also necessary to evaluate the accuracy and usefulness of μP for different processor classes.

8. REFERENCES

[1] J.A. Fisher: *Customized Instruction Sets for Embedded Processors*, Design Automation Conference (DAC), 1999

[2] A. Oraioglu, A. Veidenbaum: *Application Specific Microprocessors (Guest Editors' Introduction)*, IEEE Design & Test Magazine, Jan/Feb 2003

[3] F. Sun, S. Ravi, A. Ragnunathan, N.K. Jha: *Synthesis of Custom Processors based on Extensible Platforms*, ICCAD, 2002

[4] D. Goodwin, D. Petkov: *Automatic Generation of Application Specific Processors*, CASES, 2003

[5] H. Scharwaechter, D. Kammler, A. Wieferink et al.: *ASIP Archiecture Exploration for Efficient IPsec Encryption*, SCOPES, 2004

[6] K. Atasu, L. Pozzi, P. Ienne: *Automatic Application-Specific Instruction-Set Extensions under Microarchitectural Constraints*, DAC, 2003

[7] N. Clark, H. Zhong, S. Mahlke: *Processor Acceleration Through Automated Instruction Set Customization*, MICRO-36, 2003

[8] *SpixTools: Introduction and User's Manual*, TR 93-6, Sun Microsystems, Feb 1993

[9] VTune: http://www.intel.com/software/products/vtune

[10] D. Suresh, W. Najjar, F. Vahid et al.: *Profiling Tools for Hardware/Software Partitioning of Embedded Applications*, LCTES, 2003

[11] LISATek products: http://www.coware.com

[12] A. Nohl, G. Braun, A. Hoffmann et al.: *A Universal Technique for Fast and Flexible Instruction Set Architecture Simulation*, DAC, 2002

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

[14] M. Ravasi, M. Mattavelli: *High-level Algorithmic Complexity Evaluation for System Design*, Journal of Systems Architecture, no. 48, Elsevier, 2003

[15] L. Cai, A. Gerstlauer, D. Gajski: *Retargetable Profiling for Rapid, Early System-level Design Space Exploration*, DAC, 2004

[16] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt A. Nicolau: *EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability*, DATE, 1999

[17] R. Leupers, O. Wahlen, M. Hohenauer et al.: *An Executable Intermediate Representation for Retargetable Compilation and High-Level Code Optimization*, Int. Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS), 2003

[18] R. A. Uhlig, T. N. Mudge : *Trace-driven Memory Simulation : A Survey*, Proceedings of the ACM Computing Surveys, 1997

[19] Dinero : http://www.cs.wisc.edu/markhill/dineroiv/

[20] MPG123 Distribution : http://ftp.tu-clausthal.de/pub/unix/audio/mpg123

[21] Softfloat distribution : http://www.cs.berkeley.edu/~jhauser/arithmetic/SoftFloat.html

[22] P. Giusto, G. Martin, E. Harcourt : *Reliable Estimation of Execution Time of Embedded Software*, DATE, 2001

[23] L. Lavagno, J. R. Bammi, E. Harcourt et al: *Software Performance Estimation Strategies in a System-level Design Tool*, CODES, 2000