Selection of Instruction Set Extensions for an FPGA Embedded Processor Core

Brian F. Veale[1], John K. Antonio[1], Monte P. Tull[2], and Sean A. Jones[1]

[1]University of Oklahoma
School of Computer Science
Norman, OK 73019-6151 USA
{veale, antonio, sean.jones}@ou.edu

[2] University of Oklahoma
School of Electrical and Computer Engineering
Norman, OK 73019-1023 USA
tull@ou.edu

Abstract

A design process is presented for the selection of a set of instruction set extensions for the PowerPC 405 processor that is embedded into the Xilinx Virtex Family of FPGAs. The instruction set of the PowerPC 405 is extended by selecting additional instructions from the full 32-bit PowerPC instruction set architecture (ISA), of which the PowerPC 405 ISA is a subset. The selected instructions are supported in hardware using the reconfigurable resources of the FPGA. The proposed design process gathers execution statistics for a target application through profiling or simulation. These statistics are then used to estimate the speedup that would be achieved if selected instructions from the full PowerPC ISA are added to the ISA of the PowerPC 405 processor. An experimental study of two embedded benchmarks show significant speedup when this approach is used to extend the PowerPC 405 processor to support various floating-point operations through the use of floating-point cores developed by QinetiQ.

1. Introduction

In contrast to a general purpose microprocessor or a digital signal processor, the architecture and/or instructions implemented by an application specific instruction processor (ASIP) can be customized for a target application or application domain. In the work presented here, a Xilinx Virtex-II Pro FPGA [1] is used to implement an ASIP.

In the ASIP architecture assumed in this paper, portions of the architecture are implemented in reconfigurable hardware, which can be configured to improve the performance of a specific application or domain. A main objective of this work is to improve performance by best matching the instructions supported by the ASIP to the needs of a target application.

The hardware architecture for the assumed ASIP consists of a PowerPC 405 processor core integrated with reconfigurable resources. This architecture allows the base ISA of the PowerPC 405 to be extended to include selected instructions from the full PowerPC ISA. Support for the selected instructions is added to the PowerPC 405 by configuring the FPGA to implement their functionality. One advantage of this type of architecture is that the same hardware can be configured differently, if necessary, for different applications.

The particular focus of this paper is on a design process that uses application profiling to guide the design of a hybrid instruction set architecture (ISA) for the PowerPC 405. The concepts presented here can be applied to other hard- or soft-processor cores embedded into a reconfigurable device.

2. Related Work in Application Specific Instruction Set Processors

2.1. Related Work in ASIP Design Flows

Many design flows for ASIPs have been proposed and studied. These design flows can be classified into architectural exploration and instruction set exploration. In architectural exploration, the design engineer uses tools that guide the selection of parameters such as cache size, branch prediction strategy, and number and type of functional units. Some examples of tools that perform this type of processor customization are Sherpa [2] and BUILDABONG [3]. Such approaches allow the engineer to customize the processor to a target application by modifying the micro-architecture of the processor and thereby improve its performance for a given set of assumptions and constraints associated with the application.

Sherpa is an ASIP design framework that is used to search the design space of an ASIP. The exploration of the design space is performed by modeling the design

1-4244-0054-6/06/$20.00

problem as a set of independent optimization problems that represent specific sets of design features of the processor, such as cache size, register file size, data path size, branch prediction techniques, etc. A model for the design space is developed using a data driven analytical model or a simulation. Finally, the design parameters for each architectural feature of the processor are tuned using integer-linear programming in order to optimize the entire processor design [2].

BUILDABONG focuses on performing optimization of the architecture and compiler of the ASIP in tandem. In this approach, the user defines the base instruction set of the processor and a set of code-generation rules. A machine model for a custom compiler is extracted from the base instruction set and a user defined code-generation rule set. Next, the target application is simulated and then analyzed by an architecture exploration tool that automatically explores the architecture and compiler design spaces in order to prune the design space [3].

In instruction set exploration, basic units of the processor such as the cache and branch predictor are statically defined and the functional units of the processor can either be modified or augmented by the design engineer. The instruction set exploration design flow shown in Figure 1 customizes an ASIP for a specific application by creating customized instructions for critical portions of the application (referred to as "hotspots"). Once a set of customized instructions have been identified and implemented in hardware, the critical portions of the application can be sped up by replacing them with calls to the customized instructions [4]. This design approach allows the engineer to tailor the processor to a target application by providing special or custom instructions that will speedup the application. Examples of work in this area are MINCE [4] and AutoTie [5].

MINCE selects instructions from a pre-designed library of instructions and adds them to a processor core in order to customize the processor to a specific application. Combinational equivalence is used to ensure that the selected instructions are equivalent to the segment of application code they are chosen to implement. This approach provides an automated framework for instruction selection that effectively prunes the candidate instruction set by removing instructions that do not implement operations performed by the target application code [4].

AutoTie automatically determines the extensions required to customize a base processor to a specific application. The application is entered into the design tools as a C/C++ program. The source code of the program is then analyzed and compiled for the resulting ASIP architecture. The compilation process is used to determine what type and amount of extensions, such as register files, custom instructions, and operations, should be added to the base processor core. Performance and hardware estimation is performed to search the space of potential ASIP designs and choose the design that best matches the needs of the target application, thereby providing maximum performance [5].

2.2. Compiler Approaches

Some of the ASIP design tools and frameworks listed above include tools that generate a compiler for the processor that allows programs to be compiled towards the specific architecture of the ASIP. The type of compiler generally utilized in this area is a 'retargetable compiler' that allows customization of the compiler for the new ASIP architecture and that may permit exploration of the design space of the architecture. These compiler frameworks generally fall into one of three categories: (1) automatically generated, (2) user generated, and (3) developer generated [6].

Compilers that are automatically generated (Category 1) contain all of the information needed to work with any combination of architecture parameters within a specified architecture framework. However, these architecture frameworks generally only allow small variations in the architecture parameters. Developer generated compilers (Category 3) have the potential to support a wide set of architecture design parameters, but they require a relatively long amount of time to develop. User generated compilers (Category 2) bridge the gap between automatically and developer generated compilers, however they can take on the order of hours or days to generate [6].

Many compilers for ASIPs allow the designer to specify "hotspots" in program code that should be considered by the design tools for instruction extraction [7]. In [7], the *gcc* C compiler, part of the GNU Compiler Collection (GCC), is modified to allow the user to specify sections of code that are to be extracted as single instructions. These custom instructions can then be used elsewhere in the program. This approach assigns a single opcode for each section of user defined instructions. Disadvantages of this approach are that programs are not optimized using the extended instruction set and the programmer must be familiar with the framework used to define instructions.

In Section 3.4, a hybrid compiler, which does not have to be regenerated when the instruction set is altered, is proposed that not only allows the programmer to use a custom instruction set, but also performs some optimizations for the hybrid instruction set.

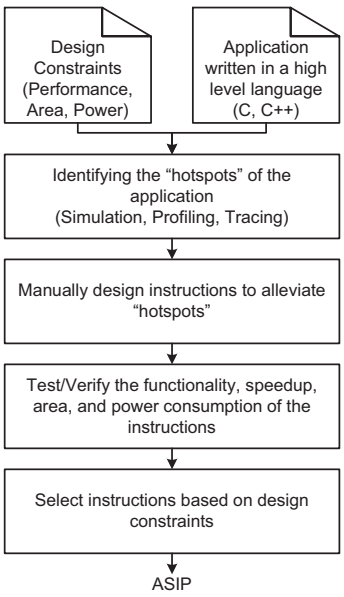

Figure 1. Instruction set exploration approach to ASIP design, derived from [4].

3. Hybrid Instruction Set Selection Process

3.1. Overview

Figure 2 illustrates a proposed design flow for selecting instructions to extend the PowerPC 405 ISA. This design flow can be applied to extend the ISA of a any base processor core (assumed here is the PowerPC 405) that implements a subset of a full ISA (assumed here is the full 32-bit PowerPC ISA such as that of the PowerPC 7400) by adding selected instructions from the full ISA to the ISA of the base processor. This allows the application engineer to create a hybrid ISA that contains some but generally not all of the instructions found in the full ISA. However, instructions are not removed from the base ISA (i.e., the hybrid ISA implements all of the instructions of the base processor's ISA).

As shown in Figure 2, the target application is compiled for both the base processor's ISA and the full ISA. Once the program has been compiled for both ISAs, the profiling step gathers statistics about the instructions that are executed by both versions of the compiled program. These statistics are used during the hybrid instruction set selection step to guide the engineer in selecting which instructions from the full ISA should be included in the hybrid ISA. Once a hybrid instruction set has been chosen, the hybrid compiler can be used to compile an application program based on the selected hybrid ISA. Sections 3.2, 3.3, and 3.4 describe the profiling, instruction selection, and compilation steps in more detail.

3.2. Application Profiling

The profiling step of the process outlined in Figure 2 can be performed using a simulator of a processor that supports the full ISA or by natively executing and tracing the execution of the compiled application on a processor that implements the full ISA. For the implementation used in this paper, the compiled application code is executed and traced using a PowerPC 7400 (PowerPC G4). The ISA of the PowerPC 405 processor, which is a subset of the full PowerPC G4 ISA, represents the base ISA. The ISA of the PowerPC 405 does not include floating-point instructions that are included in the ISA of the PowerPC G4.

A Linux-based tracing tool was developed that runs on the PowerPC G4 and can profile applications compiled for both the base processor's (PowerPC 405) ISA and the full (PowerPC G4) ISA. Because the code generated under Linux for the PowerPC 405 is compatible with the PowerPC G4, both versions of the binary are traced using a standard PowerPC G4-based machine.

The profiling tool developed for this work is based on the Linux *ptrace* [8, 9] system call. The target application is traced by the tool, one instruction at a time, using the PTRACE_SINGLESTEP mode of execution [8]. As each instruction is executed, the tool determines its mnemonic and updates how many times each instruction is executed.

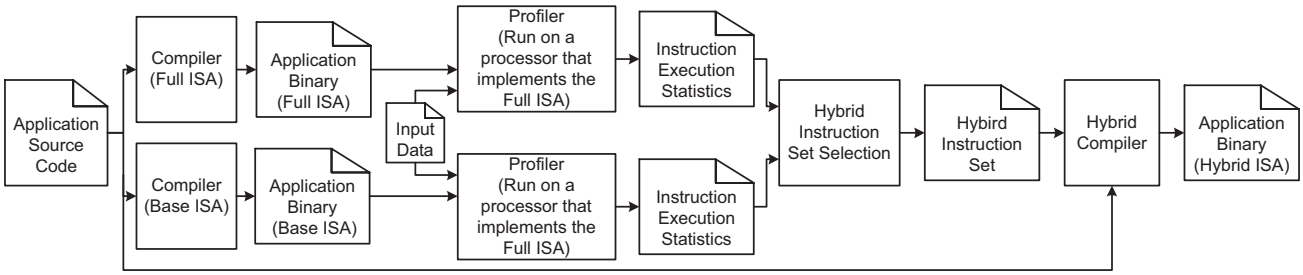

Figure 2. Hybrid instruction set selection for a Hybrid PowerPC 405. "Base ISA" refers to the ISA of the PowerPC 405 and "Full ISA" refers to the entire 32-bit PowerPC ISA such as implemented by the PowerPC 7400. This design process can also be applied to other ISA families.

The execution statistics that are gathered by the profiling tool can be combined with instruction timing values (cycles required to execute each instruction) to estimate the number of cycles required to execute the application on the base ISA, full ISA, and the chosen hybrid ISA. Due to the nature of the profiling process, the timing results are based on the assumptions of perfect caching and branch prediction. Access to a cycle accurate simulator would result in different (and generally more accurate) speedup factors than reported in this paper due to architectural features, such as cache and branch prediction policies, and would give the engineer a more precise view of expected performance of the ASIP implementation of a hybrid ISA. However, the use of a simulator would increase the amount of time required to analyze the target application and generate execution statistics to be used in the ISA selection tool. Also, the main purpose of estimating cycles required for competing ISA selections is to determine relative improvements associated with instruction selections.

Once the application is traced for the base ISA and the full ISA versions, the speedup of the full ISA version relative to the base ISA version can be computed. The profiling tool determines how many cycles are required to emulate each instruction executed from the full ISA (which are not supported by the base ISA). This is calculated by counting the number of base instructions executed by software modules used to emulate these instructions and scaling the results according to the number of clock cycles that are required to execute each base instruction executed.

After the base ISA emulation of the instructions from the full ISA is complete, the hybrid instruction set selection tool can estimate the speedup of a hybrid ISA over the base ISA. This allows the engineer to choose the set of instructions from the full ISA to be added to the base ISA, and observe the results of the choices made. This calculation of the speedup of the hybrid ISA over the base ISA is computed without re-tracing the target application.

3.3. A Framework for Automatic Instruction Set Selection

Motivated by Linear Programming models used to optimize the parameters of ASIPs using architectural exploration in [2], we propose a formal optimization model for the selection of a hybrid instruction set. The full ISA (FISA) is assumed to have N instructions that are labeled 1, 2,…, N and the base ISA (BISA) consists of the first $N_0 < N$ instructions of FISA; therefore, BISA $\subseteq$ FISA. The instructions of the FISA are supported by a collection of execution units. Unit 0 represents the base processor and supports all of the instructions in the BISA. Units 1, 2,…, U are implemented in reconfigurable hardware and collectively support the instructions in the FISA that are not in the BISA. A configuration of the ASIP includes Unit 0 plus a combination of Units 1 through U. The instructions supported by Unit 1 are labeled $N_0 + 1,\ldots, N_1$. In general, the instructions supported by Unit i are labeled $N_{i-1} + 1,\ldots, N_i$, for $i = 1, 2,\ldots, U$ (thus, $N_U = N$).

Recall from Figure 2 that the full ISA version of the application can be profiled. The optimization technique proposed in the present section requires that the number cycles required to execute each instruction j in hardware in the full ISA version of the application ($j \in$ FISA) is known. This value is denoted by n_j, where $j \in$ FISA. Also needed from the profiling process is the number of cycles required to emulate each FISA instruction j using BISA instructions, and this quantity is denoted by f_j. Additionally, $u_i \in \{0,1\}$, $i = 1, 2,\ldots, U$, indicates whether Unit i is configured in the reconfigurable hardware; if $u_i = 1$, then Unit i is configured, otherwise it is not configured.

Because there are U possible reconfigurable units and each unit is either configured or not, there are 2^U possible configurations for the ASIP architecture under consideration. Note that in this formulation, each configurable unit generally supports multiple instructions from the FISA. Associated with each possible configuration of the ASIP is the corresponding hybrid ISA (HISA) it supports.

The speedup associated with a given configuration of the ASIP relative to the BISA (i.e., the ASIP with none of the reconfigurable resources utilized) is given by Equation (1).

$$S = \frac{\sum_{j=1}^{N_0} n_j + \sum_{j=N_0+1}^{N} f_j}{\sum_{j=1}^{N_0} n_j + \sum_{i=1}^{U} \sum_{j=N_{i-1}+1}^{N_i} \left(u_i n_j + (1-u_i) f_j \right)} \qquad (1)$$

Note from Equation (1) that the "boundary conditions" of the ASIP configurations are consistent. In particular, consider first the configuration where $u_i = 1$ for all $i = 1, 2,\ldots, U$, which corresponds to the ASIP configuration in which all of the units are configured. In this case, the equation yields a speedup that corresponds to the ratio of the number of cycles required to execute instructions for the BISA divided by the cycles required assuming complete support for the FISA. The other extreme case is associated with the configuration where $u_i = 0$ for all $i = 1, 2,\ldots, U$, which corresponds to a ASIP configuration in which none of the configurable units are implemented. In this case, the formula yields a speedup of unity, as expected.

The reconfigurable resources required to implement a configuration of the ASIP are also modeled and can be

used as a constraint in the optimization of Equation 1. In this model, r_i denotes the reconfigurable resources required to implement Unit i in reconfigurable hardware where $i = 1, 2,\ldots, U$. The expression below describes the total amount of reconfigurable resources required to implement a given configuration of the ASIP.

$$\sum_{i=1}^{U} u_i r_i \qquad (2)$$

Based on the definitions presented and the expressions provided in Equations (1) and (2), an optimization problem can be formulated as follows.

Given the following four items:

1. The assumed total amount of reconfigurable resource available on the ASIP, denoted by R;
2. The amount of reconfigurable resource required for each reconfigurable unit under consideration, denoted as r_i, $i = 1, 2,\ldots, U$;
3. The number cycles required by each instruction in the FISA to be executed in hardware, denoted by n_j, $j = 1, 2,\ldots, N$;
4. The total number of cycles required to execute the BISA instructions used to emulate each instruction in the FISA, denoted by f_j, for all $j = 1,\ldots, N$;

$$\max_{\substack{u_i \in \{0,1\} \\ i \in \{1,2,\ldots,U\}}} \{S\}$$

$$\text{subject to } \sum_{i=1}^{U} u_i r_i \le R$$

The dual problem of minimizing the required amount of reconfigurable resource subject to a lower-bound constraint on the speedup can also be formulated.

Note that there are 2^U possible configurations for the ASIP. Thus, an exhaustive search approach for solving the formulated optimization problem requires up to 2^U evaluations of the formula for S given in Equation (1). Typically the value of U will be relatively small (less than ten), thus evaluating the speedup for all combinations in an off-line design process is reasonable.

3.4. Hybrid Compiler

The final step of the design flow shown in Figure 2 involves a compiler that generates machine code based on the selected hybrid ISA. The compiler assumed in the proposed design flow differs from traditional compilers used with ASIPs that are based on instruction set exploration. Typically, compilers for instruction set exploration operate with program source code that has been annotated to use new instructions generated by the design tools being used or that have been provided by the programmer.

The hybrid ISA for the ASIP considered in this paper is a subset of an existing full ISA. Furthermore, the hybrid ISA includes all of the instructions of an existing base ISA, which is also a subset of the full ISA.

A compiler targeting the full ISA can be modified to compile towards an appropriate subset of the full ISA. Thus, it is possible to modify an existing full ISA compiler to create code associated with a hybrid ISA that is a subset of the full ISA. Because no instructions have been added, the compiler can still perform all of the machine independent (i.e., intermediate form) code optimizations and then use emulation routines as necessary to create the final hybrid code from the intermediate form. This requires the availability of a library that contains the necessary emulation routines to support the operations not directly supported by the hybrid ISA. Employment of emulation routines is the same approach used when a compiler generates code for a processor lacking a floating-point processing unit (FPU), e.g., the compiler for the PowerPC 405.

We have developed a prototype hybrid compiler that effectively merges compilers for the PowerPC 7400 (full ISA) and the PowerPC 405 (base ISA) to generate code based on a selected hybrid ISA that is a subset of the full ISA. This hybrid complier takes the desired hybrid ISA as an input. Because the data paths, registers, and other architectural components of the architecture, except for configurable execution units, are fixed, the hybrid compiler does not have to be regenerated or recompiled when the instruction set of the hybrid processor is modified. This approach enables the compiler to compile for any subset of the full ISA and still perform optimizations on the resulting intermediate code and the final machine code.

For example, the experimental studies of Section 4 can be supported by combining the PowerPC 405 found in some members of the Xilinx Virtex family of FPGAs with floating-point soft-cores provided by QinetiQ [10]. The hybrid ISA consists of the base PowerPC 405 ISA extended with floating-point instructions from the full PowerPC ISA. QinetiQ provides a modified version of the *gcc* C compiler [11] that implements a system similar to the hybrid compiler described in this section.

The modifications made to the *gcc* C compiler by QinetiQ include a *–mfpu* command line switch, through which the developer is able to select what level of floating-point support the compiler is to add to the instruction set used during compilation. The levels of floating-point support available in QinetiQ's *gcc* compiler include: (1) no floating-point support, (2) basic floating-point support, (3) basic floating-point support with division, (4) basic floating-point support with square root, and (5) full floating-point support [11]. While these levels of support reflect the capabilities of QinetiQ's line of floating-point cores, the modifications made to the compiler are not vendor specific. It is possible to modify

the compiler to support other soft floating-point cores for the PowerPC405.

The compiler provided by QinetiQ provides a medium scale of granularity relative to the control of the floating-point support provided; i.e., the designer can choose one of several configurations that add groups of multiple floating-point instructions to the ISA. Our proposed design process allows for an even finer level of granularity in which the addition of single instructions from the full ISA to the base ISA can be made. Additionally, this approach can be extended to apply to instructions other than just floating-point instructions.

4. Experimental Results

In this section, the base PowerPC 405 ISA is extended by selecting additional instructions from the full PowerPC G4 ISA using the hybrid instruction set selection process of Section 3.3. Tools that support all of the steps of Figure 2 have been developed. The study presented here focuses on the speedup achieved when extending the base PowerPC 405 ISA to include selected floating-point instructions from the full PowerPC G4 ISA. Reconfigurable resource requirements presented are estimated assuming the use of the Quixilica floating-point execution unit cores from QinetiQ [10].

In order to determine how many cycles are required by the floating-point instructions used in the target application, the profiling tool discussed in Section 3.2, captures data operands from the floating-point registers that are used by the floating-point instructions of the full ISA while the full ISA version of the application is being traced. These collected register values are then provided as input to the emulation modules, which use only base ISA (integer-based) instructions to support the floating-point operations associated with the full ISA. By profiling the emulation code in this way, determination of the number of cycles required is based on the same data values that were used by the corresponding floating-point instructions associated with the full ISA version of the application. This is important because the number of cycles (required to emulate a floating-point instruction) is dependent on the values of the input data operands to these emulation modules.

Because the Quixilica cores and PowerPC 405 run at different clock rates, the latency of the instructions supported by the core are normalized to the speed of the PowerPC 405 (which is assumed to run at 300 MHz as defined in [12]). Additionally, the latencies account for communication overhead between the PowerPC 405 and the Quixilica cores, which are assumed to be connected to the Processor Local Bus (PLB) of the PowerPC 405 that is available to reconfigurable resources of the FPGA [12].

The benchmarks studied here are the Basicmath and Susan benchmarks from the Automotive and Industrial Control category of the MiBench set of embedded benchmarks [13]. The purpose of the Basicmath benchmark is to exercise the processor to see how well it can perform mathematical operations. The Susan benchmark is an application that is used to detect corners and edges in images [13].

Figures 3 and 4 show the estimated speedup of the Basicmath and Susan benchmarks versus specific floating-point instructions that can be added to the base PowerPC 405 ISA. The maximum speedup for Basicmath is 6.9 and 3.5 for Susan. Figures 5 and 6 show the number of slices (a measure of the amount of utilized reconfigurable resource for the Virtex-family of parts) required to configure hardware support for these extra instructions in a Virtex-II Pro FPGA; these values are derived from [10].

Note that the speedup and number of slices required for the selected instructions is cumulative, e.g., the speedup shown for the *fsub* instruction in Figure 3 assumes that the hybrid instruction set also includes the instructions *fadd*, *fmul*, and *fmadd*. Since the Quixilica floating-point execution unit cores provide support for a subset of the floating-point instructions present in the full PowerPC G4 ISA, the model used in this study only allows the instructions supported by Quixilica to be added to the base ISA. The remaining instructions, *fabs*, *fcmpu*, *fctiwz*, *fmr*, *fnabs*, *fneg*, and *frsp*, which are not shown in Figures 3 - 6, must be performed using emulation libraries. Also, the model assumes that floating-point load and store instructions are supported in hardware.

The order in which instructions are added to the hybrid ISA affects the shape of the speedup curve observed for a particular benchmark. Furthermore, observe that in the speedup results for both Basicmath and Susan there is a point at which no increase in speedup occurs as more instructions are selected. This occurs because these instructions are not executed (frequently) in the benchmark.

Figures 5 and 6 show how multiple instructions can be added to the instruction set together without requiring extra hardware resources. This benefit comes as a result of adding functional units to the base processor as needed instead of modifying the existing hardware to accommodate the new instruction(s). In this unit-based approach, certain instructions become intrinsically supported when an instruction associated with the same unit is added to the instruction set.

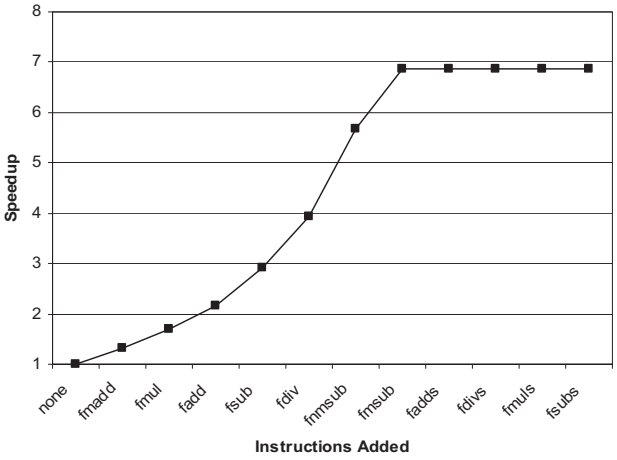

Figure 3. Speedup for Basicmath where floating-point instructions are added to the hybrid ISA according to the number of cycles required to execute the base ISA version due to emulation.

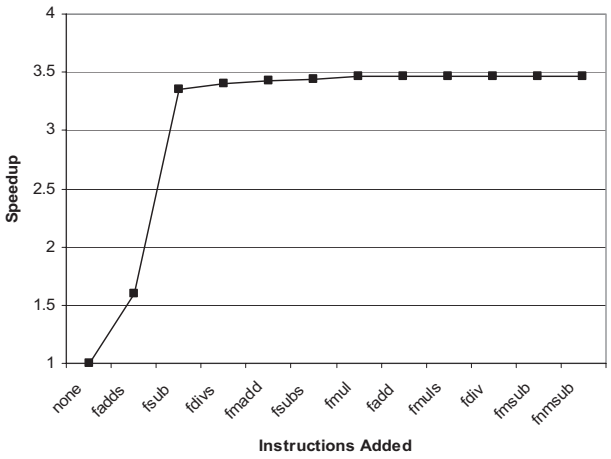

Figure 4. Speedup for Susan where floating-point instructions are added to the hybrid ISA according to the number of cycles required to execute the base ISA version due to emulation.

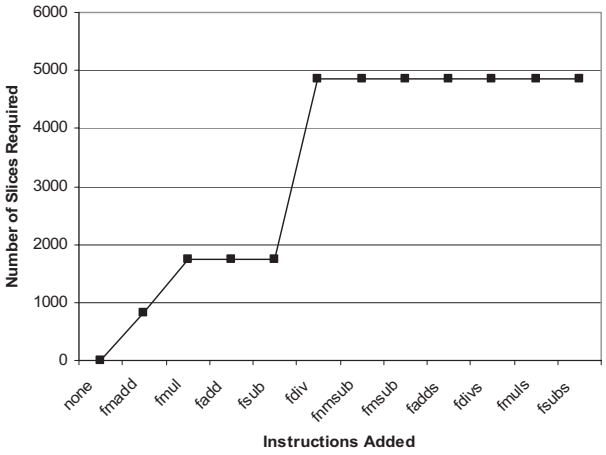

Figure 5. Amount of reconfigurable resources required to support Basicmath for the instructions shown in Figure 3.

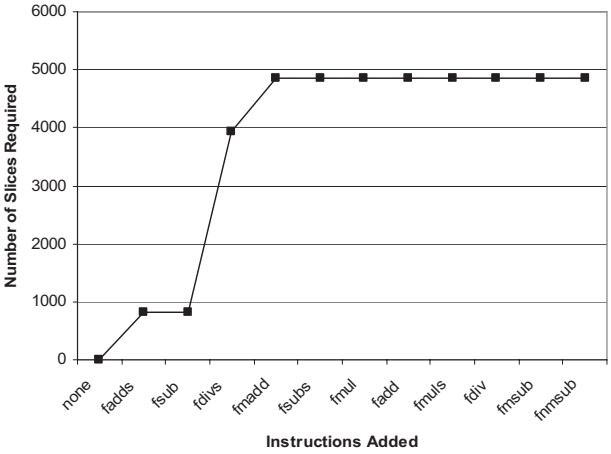

Figure 6. Amount of reconfigurable resources required to support Susan for the instructions shown in Figure 4.

Table 1 presents the result of enumerating all possible configurations for the Basicmath application when only three double-precision floating-point units (provided as part of the Quixilica floating-point cores) are considered for inclusion in the ASIP. Note that Quixilica [10] provides both single and double precision versions of the floating-point units; however, Basicmath [13] only uses double precision floating-point instructions. As discussed earlier, certain floating-point instructions are not supported by the Quixilica floating-point execution unit cores and are always emulated using emulation libraries.

Table 1 illustrates that some configurations of the ASIP that require more reconfigurable resource than others do not always deliver better performance in terms of speedup. For example, compare the configurations where a floating-point adder and divider are used versus the configuration where a floating-point adder and multiplier are used.

Table 1. The amount of reconfigurable hardware required and the resulting speedup for Basicmath when different combinations of the Quixilica double-precision floating-point cores are used to augment the PowerPC 405 BISA.

FP Add	FP Multiply	FP Divide	# of Slices	Speedup
0	0	0	0	1.0
0	0	1	3127	1.1
0	1	0	923	1.3
0	1	1	4050	1.5
1	0	0	815	1.3
1	0	1	3942	1.5
1	1	0	1738	4.5
1	1	1	4865	6.9

5. Conclusions

A process for selecting an ISA for a configurable ASIP is introduced. The approach assumes the ASIP is capable of supporting a range of possible ISAs. Each of these ISAs represents a hybrid combination of two extreme ISAs referred to as the base ISA and the full ISA. The ISA selection approach is evaluated in the context of an existing commercially available product that can function as an ASIP. Future work includes finalizing the initial development of a hybrid compiler, which is necessary to complete the implementation of the proposed process, extension of the analytical model, and development of a prototype ASIP using a Xilinx Virtex FPGA.

6. References

[1] Telikepalli, A., "*Virtex-II Pro FPGAs: The Platform for Programmable Systems has Arrived*," XCell Journal, http://xilinx.com/xcell, Mar. 2002, No. 42.

[2] Sherwood, T., Oskin, M., and Calder, B., "Balancing Design Options with Sherpa," *Proceedings of the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, Sep. 2004, pp. 57-68.

[3] Fischer, D., Teich, J., Thies, M., et al, "Efficient Architecture/Compiler Co-Exploration for ASIPs," *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, Oct. 2002, pp. 27-34.

[4] Cheung, N., Parameswaran, S., Henkel, J., et al, "MINCE: Matching Instructions using Combinational Equivalence for Extensible Processor," *Proceedings of the 2004 Design Automation and Test in Europe Conference and Exhibition*, vol. 2, Feb. 2004, pp. 1020-1025.

[5] Goodwin, D., and Petkov, D., "Automatic Generation of Application Specific Processors," *Proceedings of the 2003 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, Oct. 2003, pp. 137-147.

[6] Liem, C., *Retargetable Compilers for Embedded Core Processors: Methods and Experiences in Industrial Applications*, Kluwer Academic Publishers, Boston, MA, 1997.

[7] La Rosa, A., Lavagno, L., and Passerone, C., "A Software Development Tool Chain for a Reconfigurable Processor," *Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, Nov. 2001, pp. 93-98.

[8] Sandeep, S., "Process Tracing using Ptrace," *Linux Gazette*, Iss. 81, http://www.linuxgazette.com/node/1333, August 2002.

[9] Bovet, D. and Cesati, M., *Understanding the Linux Kernel*, Second Edition, O'Reilly and Associates, Cambridge, Massachusetts, 2002.

[10] *Quixilica Floating Point Cores: IEEE-754 Compliant Variable Wordlength Floating Point Arithmetic Cores for Xilinx Virtex and Spartan FPGA Families*, Datasheet QINETIQ/S&E/APC/TDS030105, Issue 3, QinetiQ, Ltd., http://www.qinetiq.com/home_rtes/quixilica_products/firmware_cores.html, June 2004.

[11] *GCC Compiler for the Quixilica Floating Point Unit for the Xilinx Virtex-II Pro FPGA*, QinetiQ, Ltd., Email: support@quixilica.com, Aug. 2005.

[12] *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet*, DS083, version 4.3, Xilinx, Inc., http://www.xilinx.com/bvdocs/publications/ds083.pdf, June 2005.

[13] Guthaus, M. R., Ringenberg, D. E., Austin, T. M., et al, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," *Proceedings of the 4th Annual IEEE Workshop on Workload Characterization*, Dec. 2001, pp. 3-14.