Compiler-directed Customization of ASIP Cores

T Vinod Kumar Gupta
University of Maryland
College Park, MD
tvinod@eng.umd.edu

Roberto E. Ko
Cornell University
Ithaca, NY
ek56@cornell.edu

Rajeev Barua
University of Maryland
College Park, MD
barua@eng.umd.edu

ABSTRACT

This paper presents an automatic method to customize embedded application-specific instruction processors (ASIPs) based on compiler analysis. ASIPs, also known as embedded soft cores, allow certain hardware parameters in the processor to be customized for a specific application domain. They offer low design cost as they use pre-designed and verified components. Our design goal is choosing parameter values for fastest runtime within a given silicon area budget for a particular application set. Present-day technologies for choosing parameter values rely on exhaustive simulation of the application set on all possible combinations of parameter values – a time-consuming and non-scalable procedure. We propose a compiler-based method that automatically derives the optimal values of parameters without simulating any configuration. Further, we expand the space of parameters that can be changed from the limited set today, and evaluate the importance of each. Results show that for our benchmarks, the runtimes for different configurations are predicted with an average error of 2.5%. In the two area constrained customization problem we evaluate, our method is able to recommend the same configuration that is recommended by brute force exhaustive simulation.[1]

Keywords: customization, embedded, soft cores, ASIP

1. INTRODUCTION

A major challenge facing embedded system designers is the tedious and expensive design process of custom chips. Advances in hardware-software co-design have yielded high-performance custom designs, yet the design of custom chips remains expensive and time-consuming. Consequently, many systems use general-purpose embedded processors despite their much inferior performance and power characteristics [7].

A recent innovation are embedded soft cores, which are intermediate between custom and general-purpose designs. Soft cores are general-purpose processors that have parameterizable components instead of a fixed design. Since customization is per application-set, they are also called application specific instruction processors (ASIPs). Examples include those from Tensilica [14], ArcCores [1] and HP [2, 3]. Application-specific configuration yields better performance than general purpose chips, while pre-designed and verified components yield lower cost than custom chips. The trend towards implementing embedded systems as a system-on-a-chip (SOC) has increased the attraction of soft cores as the SOC fabrication is often done on a per-application basis anyway. Soft core technology is in its infancy today – tools to help choose parameter values rely on exhaustive search, and the space of customizable parameters remains small.

Customization of soft cores on a per application-set basis may yield benefits because not all applications use different CPU resources equally. For example, multiply-accumulate (MAC) functional units are profitable if the application set has many incidences of computations such as $(a \times b) + c$ without the $(a \times b)$ value being used elsewhere. For other applications which have few such computations but (say) many parallel memory operations, it may be profitable to provide dual-ported memory instead. As the silicon area available may be limited due to cost and power constraints, having all possible enhancements such as MAC and dual-ported memory may not be possible. In such a scenario, choosing the CPU design on an application-specific basis will yield lower runtime and/or lower cost and power consumption.

Although ASIPs yield many advantages over hard cores, their customization poses serious challenges. Existing methods usually require the user to compile and run the target application on all possible configuration of the ASIP. This exhaustive search is time-consuming because the number of configurations is exponential in the number of parameter values; further each configuration requires simulation with potentially large data sets. For parameters such as memory size that can take on a large number of values, the number of configurations grows very rapidly. A time-consuming search is undesirable for three reasons. First, a long design time increases the time-to-market and cost for a new design; a quick time-to-market is critical for rapid innovation. Second, for many low-volume chips, a high custom design cost forces the use of general-purpose chips. Third, a design time that exponentially grows with the number of parameters acts as a disincentive for increasing the number of customizable parameters – this has probably already happened, as seen in the small number of parameters in commercial designs [14, 1, 2]. A larger number of customizable parameters provides more opportunities for the design to closely match the ideal requirements of the application set, thus increasing performance within the same silicon area.

In this paper, we present a method by which the optimal values of architectural parameters for a given application set are automatically derived; and further, more customizable parameters are pro-

[1] This research is funded in part by an NSF CAREER award to Barua.

CODES'02, May 6-8, 2002, Estes Park, Colorado, USA.
Copyright 2002 ACM 1-58113-542-4/02/0005...$5.00.

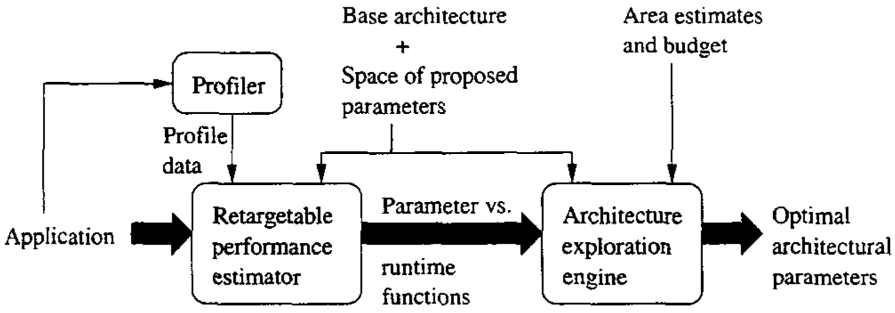

Base architecture
+
Space of proposed parameters
Area estimates and budget
Profiler
Profile data
Application
Retargetable performance estimator
Parameter vs. runtime functions
Architecture exploration engine
Optimal architectural parameters

Figure 1: Methodology for Core Customization

posed. The optimality goal is maximum application performance within a given area budget. Our method predicts the best configuration of the embedded core without actually running on any configuration. The method analyzes the application using a compiler framework that uses static and profile information to deduce the best configuration using a proposed mathematical model.

Our method works as follows. First, to be able to predict performance of different configurations of the architecture at compile time, we use a simplified processor description language to describe the processor. The description language is expressed as a context-free grammar that can describe most embedded VLIW and single-issue processors, and the values of the parameters that are to be customized. Second, the description is provided to the instruction-scheduler in our compiler that derives a schedule, and thus a performance estimate, for the application on the described processor. Third, by predicting performance for configurations that have only one CPU enhancement (parameter) present, we show a method by which the performance of other configurations can be inferred by using statistical data on the inter-dependence of the given parameters.

The approach aims to improves upon the state-of-the-art in two ways. First, the optimal configuration is proposed to be derived, for the first time, without exhaustive simulation-based search of the design space. The proposed mathematical model captures the inter-dependence between the impact of different parameters on program execution, using *dependence constants*. These constants preclude the need to exhaustively simulate all combinations of parameter values. Second, the parameters evaluated in this research are novel in their nature, as they have never been evaluated in this context. Overall, this rapid customization will allow low-volume chips to be customized - many of these are forced to use general-purpose chips today to hold down costs.

In this paper, we implement and evaluate our methodology to verify the correctness of its predictions. We use our methodology to evaluate four parameters, namely, presence or absence of mac functional unit, presence or absence of an hardware floating point unit, need for a single-ported or dual-ported memory and the choice between a non-pipelined and a pipelined memory unit. The architecture platform used is the Philips TriMedia VLIW processor. Our methodology predicts applications performance with an average error of under 2.5% between predicted and actual performance. In the area constrained customization problem we evaluate, our methodology is able to recommend the exact same configuration that is recommended by a brute force exhaustive simulation methodology.

2. RELATED WORK

Prior work in ASIP customization is restricted to approaches that either evaluate configurations exhaustively or evaluate certain parameters only in isolation. Further, cost-area-based analysis is absent in most related work.

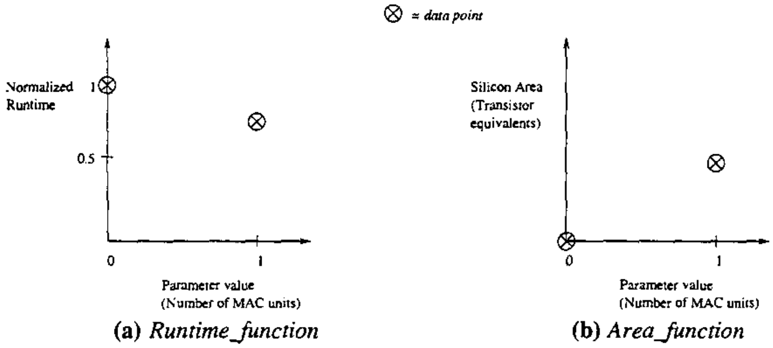
Commercial soft cores include those by Tensilica [14] and ARC Cores [1] corporations. They offer the user the ability to select the instruction set mix of the processor, its addressing modes and sizes of internal memory banks. They supply a tool to the client that, for each of these parameters, estimates the impact on area of varying that parameter. Experienced designers at the client decide optimal configuration based on their experience or by experimentally evaluating possible configurations using a simulator.

Gong et. al. [6] evaluate architectural features such as machine parallelism, number of buses and their connectivity and memory ports using their performance evaluator. The methodology computes speedups by various parameters, but does not compute speedups due to combination of parameters. Ghazal et. al. [5] predicts runtimes of applications that can take advantage of advanced processor features and compiler optimizations such as optimized special operations, memory addressing support, control-flow support, and loop-level optimization support. Gupta et. al. [8] analyze target applications to extract its characteristics. The characteristics, in turn, give an indication of some of the architectural features of suitable processor. But the methodology does not quantify the impact of each derived feature. Moreover, the performance estimation stage of the methodology requires exhaustive simulations.

Work by Kuulusa et. al. [10], Hebert et. al. [9] and Shackleford et. al. [13] build software tools to do architecture parameter space exploration by exhaustive search. They also evaluate extensions to instruction sets. The work on *Custom-Fit Processors* [3, 2] also uses exhaustive search, but targets a a VLIW architecture framework in which several characteristics can be changed: memory sizes, registers sizes, kinds and latencies of functional units and clustered machines. speedup/Cost graphs are derived for all possible combinations yielding pareto points.

3. APPROACH

Our approach customizes an existing base processor with enhanced architectural features (parameters) that are optimized for a group of target applications. The design flow of the methodology is shown in figure 1. Given an applicatiun code and base architecture configuration, the performance analyze; estimates the runtime of the application with different architectural parameters. The results of this step, area estimates of each parameter, and overall area budget are used by a architecture-exploration engine that chooses optimal parameter values. The following sections explain the process in detail.



(a) *Runtime_function* (b) *Area_function*

Figure 2: Example curves for MAC unit parameter. (a) shows how performance (Normalized runtime) may vary with the value of the parameter, which is the number of MAC units, for some application. Runtimes are estimated by profile-augmented compiler analysis. (b) shows how the silicon area varies with the parameter value. The areas of pre-fabricated blocks is known to the processor vendor.

3.1 Problem formulation

Let n be the number of customizable parameters. The method solves for $p_1, \ldots, p_n$, where p_i represents the numerical value of the i^{th} parameter. For example, let the i^{th} parameter be the presence or absence of a multiply-accumulate (MAC) unit. A MAC unit performs a multiply and a dependent add in the same time as an ordinary multiply. The number of MACs, represented by p_i, is either 0 or 1 (it can be more for VLIWs). Without loss of generality, assume the base architecture has $p_i = base_i = 0$. Our goal is to find the values of p_i that maximize performance within a certain silicon area. Our approach has three steps. Steps 1 and 2 are performed for each of the n parameters; step 3 combines the results.

Step 1: Derive runtime vs. parameter curves In this step, a performance vs. parameter analyzer is derived for each parameter, and used to plot the predicted performance of the application set as a function of the parameter value p_i. This is best explained through an example. Figure 2(a) shows how the runtime vs. parameter curve may look like for some application set, for the parameter tracking the presence or absence of a multiply-accumulate (MAC) unit. The function thus obtained is called $Runtime\_function(p_i)$, and assumes that $p_j = 0$ for all $j \neq i$. The $Runtime\_function$ is a normalized function which is simply the ratio of the runtime with parameter value of p_i and base case runtime (when the parameter value is the base value of that parameter). The runtime in each case is obtained by profiling – the profile-collected basic block frequencies are multiplied by our VLIW-scheduler-predicted runtime of that basic block[2]. When the MAC unit is present, however, a lower runtime is predicted for basic blocks that use the MAC unit. The lower prediction is obtained by re-compiling the application with the compiler looking for multiply-accumulate patterns in the data-flow graph (DFG) of the application, and recomputing the runtime with MAC instructions.

Our approach is computationally more tractable than existing methods as it re-compiles for every parameter value, rather than re-running for every combination of parameter values. For one, re-compiling the application is usually cheaper than re-running it, especially for large data sets. More importantly, the number of parameter values grows linearly with itself, while the number of combinations of parameter values grows exponentially.

Step 2: Obtain area vs. parameter curves Here we obtain curves plotting the parameter values vs. the additional area required for that parameter value, beyond when the parameter is not used. Figure 2(b) shows how the area vs. parameter curve may look like for the application set in consideration, for the MAC unit parameter. The area for number of MACs = p_i = 0 is, by definition, zero. The area for number of MACs = 1 is the area of a MAC unit in terms of gate equivalents. The area required for pre-fabricated units used as parameters is, of course, known to the vendor of embedded soft core. The function obtained in this step is called $Area\_function(p_i)$. The $Runtime\_function(p_i)$ and $Area\_function(p_i)$ are maintained as a set of points; there is no need to try to curve-fit them into a symbolic form.

Step 3: Architecture-exploration engine chooses optimal parameter values To obtain the best performance, the different parameters are traded-off to obtain the best usage of silicon area for that application. Finding optimal parameter values is a constrained optimization problem, defined by a set of variables, a set of constraints and an objective function to be minimized. The variables are the p_i ($i \in [1, n]$). The constraint is that sum of the area functions for all parameters is within the given fixed area budget:

$$\sum_{i=1}^{n} Area\_function(p_i) \leq AREA\_BUDGET \qquad (1)$$

The objective function to be minimized is the predicted overall runtime of the application set as a function of the parameter values $p_1, \ldots, p_n$. Hypothetically, if we were to make the simplifying assumption that the performance impact of each parameter were independent of the others, then the objective function would be simply the product of the normalized runtime functions for each parameter ($\prod$ represents series product):

$$Pred\_Runtime\_function = \prod_{i=1}^{n} Runtime\_function(p_i). \qquad (2)$$

The difficulty is that the parameters are not all independent, making the formula in (2) inaccurate. For example, increasing the value of p_i may either increase or decrease the gain from increasing another parameter p_j, as the pair may be synergistic or conflicting, respectively. This work proposes and investigates a method by which the interdependence between parameters can be accounted for. The method incorporates a heuristic $Dependence\_constant$ for

[2] Adding instruction latencies yields an accurate runtime for a basic block as virtually all embedded processors are single-issue machines or VLIWs, not dynamic issue superscalars. Further, if there is more than one application to be run on the desired processor, their profile data is combined and weighted by their relative execution frequencies.

every combination of parameters, that adjusts for the gain for that combination. In particular, the objective function in formula 2, computing predicted overall normalized runtime, could be adjusted as follows:

$$Pred\_Runtime\_function = \frac{\prod_{i=1}^{n} Runtime\_function(p_i)}{Dependence\_constant(p_1, \cdots p_n)} \quad (3)$$

The *Dependence_constant* is a value that adjusts the gain to take into account dependences between parameter values. For example, if a particular combination of $p_1, \ldots, p_n$ is synergistic, the *Dependence_constant* will be greater than 1 for that combination; and less than 1 if conflicting. To collect values for this set of constants, the intuition is that if the parameters are interdependent in a certain way for a standard set of benchmarks, then perhaps they are interdependent in a similar way for the application set under consideration. Different benchmark sets may be used from different domains, such as signal-processing code, micro-controller code and communication protocol code, to more closely match the current application set. The *Dependence_constant* for a given combination of values $p_1, \cdots, p_n$ is a constant value obtained from exhaustively simulating the standard set of benchmarks, *not* the given application set. The constant values are collected one-time only by the vendor, not per design. They are computed as follows: for each combination $p_1, \ldots, p_n$, the actual normalized runtime is compared with the predicted runtime in formula 2. The constant factor is the ratio of the two. Mathematically,

$$Dependence\_constant(p_1, \cdots, p_n) =$$

$$\begin{cases} 1 & : if \forall i, p_i = base_i \\ 1 & : if \exists j, p_j \neq base_j, \\ & \quad \forall i \neq j, p_i = base_i \\ \frac{(\prod_{i=1}^{n} Runtime\_function_i(p_i))}{Actual\_normalized\_runtime(p_1, \cdots, p_n)} & : otherwise \end{cases} \quad (4)$$

This heuristic to take into account interdependence is not exact. Nevertheless, it has the potential to be close enough to accurate to allow the exploration engine to choose a good combination of parameter values.

3.2 Evaluated Parameter List

Listed below are four new parameters we have identified that we believe have the potential to be profitably varied for better application-specific performance.

Presence vs. absence of MAC unit This tracks the presence or absence of a multiply-accumulate (MAC) unit. A MAC unit performs a multiply and a dependent add in the same time as a multiply. Its value depends upon how frequently the application has MAC computations.

Hardware vs. Software floating point Floating point operations can be done in hardware or simulated in software. Hardware's advantage is speed; but software saves on the often-expensive area of floating point units.

Multiported Memory Multiported memory allows multiple memory operations (loads and stores) simultaneously. For VLIWs, this has the advantage of speeding up memory critical applications but at a higher cost.

Pipelined vs. non-pipelined Memory Unit Pipelined memory units allow memory operations to issue to a memory unit before it has completed the previous memory instructions issued to that unit.

Apart from the above four parameters that have been evaluated in this paper, a whole range of new parameters can also be considered. These may include register file size, number of architectural clusters, number and nature of functional units, presence of an address generation unit.

4. ARCHITECTURE MODEL

Our approach estimates the runtime of applications on the base architecture with parameterized components. To accurately estimate runtime in a retargetable manner, the performance estimator must model the architecture in some description language. We choose an architecture representation that can express most embedded VLIW processors. Based on the one used by Gupta et. al. [8], it incorporates the different types of functional units that are present in the datapath, along with their associated operations, corresponding latencies and delays. The model also captures the constraints of typical VLIW cores in terms of operation slots and slot restrictions. Finally, one can specify the number of registers in the core, number of write buses, number of ports in the memory, and the delay of branch operations.

The generic model provides the freedom to configure the architecture with varying levels of parallelism to offer. Also the nature, kind and number of functional units can be altered to gauge their effect on application behavior.

5. RETARGETABLE PERFORMANCE ESTIMATOR

To compute the runtime-function estimates we need a compiler based performance estimator. The estimator should be retargetable to account for any input architecture specified by the architecture description language. Once configured for the architecture, the estimator should then return an estimate of the runtime for any given application on that architecture. The estimator consists of three new components:a profiler module, a data flow graph (DFG) builder and a fine-grain scheduler; each is described below.

Profiler module The profiler module computes the execution frequencies of each basic block. To do this, it declares one new global variable for each basic block that keeps count of the number of times that block is executed. Each variable is initialized to zero in the main function. At the start of each basic block, the variable associated with it is incremented by one by a compiler-inserted instruction. The execution counts are available when the program exits.

Data flow graph builder To schedule the application code, a data flow graph is needed. The data flow graph builder takes a basic block as input and outputs a directed acyclic graph (DAG) that captures all types of dependences between the operations. The vertices in the graph correspond to each operation in the basic block and an edge between two vertices represent the dependence between them.

Fine-grain scheduler The scheduler takes as input the DFGs for basic blocks, the target architecture description as specified in section 4, and the application's profile data. It schedules each basic block on the architecture to derive the execution time for each basic block. Next, it combines this with the execution frequency of each block to estimate the overall runtime of the application on the given architecture.

The scheduler uses list scheduling [12] to schedule each basic block. List scheduling is a greedy method that chooses the next instruction to schedule among those ready to fire in the DAG in the order of their *priority*. The priority for each instruction depends on the length of the critical path from that instruction to the end of the basic block – the longer the path, the higher the priority. Chosen instructions are scheduled in the first available slot in the VLIW schedule. The priorities, and hence the schedule, is machine-dependent as the length of the critical paths depends on

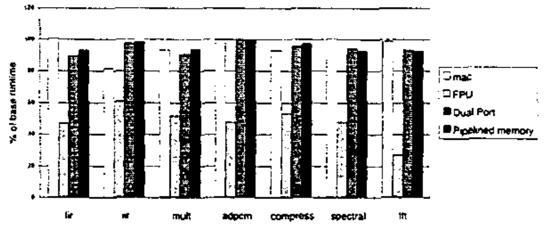


Figure 3: Performance effect of INDIVIDUAL parameters. The figure shows the runtime_functions for each benchmark and application for each of the parameters chosen, namely, mac, FPU, dual-ported memory and pipelined memory.

the latencies of the instructions on that architecture.

The scheduler also takes into account the special architectural parameters being considered for configuring the soft core. The four parameters evaluated in this paper are incorporated into the scheduler as follows. First, the *mac* operations are detected in the DFG, and scheduled as instructions in architecture configurations that include a MAC unit in hardware. A *mac* instruction is used only if the multiply operation has only one successor, otherwise it is not profitable. Second, floating point operations in the DFG are scheduled as instructions if hardware floating point is available: otherwise calls to equivalent software routines are inserted for each floating point operation. Third, depending on the number of memory ports specified in the architecture description, usually specified by available VLIW slots for memory instructions, the scheduler incorporates constraints on the number of memory ports correctly. Fourth, based on the presence of pipelined memory units, the scheduler frees up the memory units one cycle after it has issued a memory operation.

6. CASE ANALYSIS AND RESULTS

Architecture core For our baseline DSP architecture, we model a Philips TriMedia processor. This is a 5-wide VLIW design with a load/store architecture. All logic and arithmetic operations are performed through the register file. The processor has 27 functional units including those for floating point arithmetic. It also has a dual ported memory. The memory functional unit has a 3 cycle delay. Perfect branch prediction is assumed for the studies. Experiments are conducted to evaluate the effect of changing parameters on application performance.

Benchmarks and Applications Programs from the UTDSP benchmark suite [11] are randomly divided into two sets of four 'benchmarks' and three 'applications'. The benchmarks are used to predict the performance of the applications. The benchmarks include *fir, iir, mult* and *adpcm*. The applications are *compress, spectral* and *fft*.

6.1 Computing Dependence Constants

The dependence constants between architectural parameters are obtained by running exhaustive simulations over all benchmarks. The constants are computed using equation 4 in section 3, which involves exhaustive simulation of the benchmarks on all possible parameter combinations to compute the *Actual_normalized_runtime* for each combination. Thereafter, for other applications, the average dependence constants are applied to predict its performance for various configurations. Hence there is no need to carry out exhaustive simulations for the other applications. If the predicted performance is close to the actual measured performance, then the methodology is verified.

Actual individual parameter effect Figure 3 shows the improvement in performance of the benchmarks for each parameter individually. The bars for each benchmark show its cycle count as a percentage of the cycle count on a base architecture, i.e. *runtime_functions*. It is seen that a given parameter can have varying impact on different benchmarks.

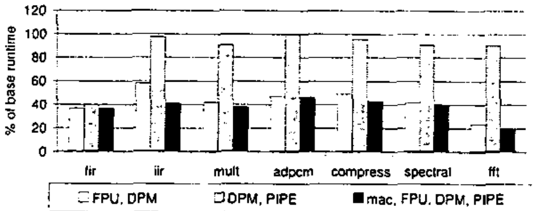


Figure 4: Performance effect of selected COMBINED parameters. The figure shows the runtime_functions for each benchmark and application for selected combinations of the parameters chosen. DPM refers to dual-ported memory and PIPE refers to pipelined memory

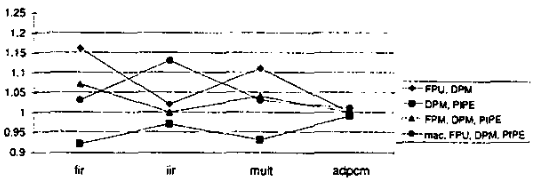


Figure 5: Dependence constants for selected combined parameters.

Actual Combined Parameter Effect Figure 4 shows the improvement in performance for selected combinations of two or more parameters in terms of *runtime_functions*. For four parameters, there are eleven such combinations.

Deriving Dependence Constants Figure 5 shows the values of each dependence constant for the different benchmarks. They are derived from the values of actual individual and combined parameter *runtime_functions*, using equation 4. The average dependence constant ranged from 0.955 for (mac, dual-ported memory, pipelined memory) to 1.12 for (mac, FPU, dual-ported memory).

It is important to note that a dependence constant that is close to one does not imply that the corresponding set of parameters are independent. The reason is that the set of parameters the constant is modelling impact only a fraction of the instructions in a program. As an example, consider the two parameters dual-ported memory and pipelined memory, both of which target memory parallelism. Our results did show that they are highly dependent – the gain from both is almost the same (7.3%) on average as from each individually (6.1%). Thus the dependence constant from equation 4 is (0.939 * 0.939)/(0.927) = 0.95. This is close to one despite the parameters being highly dependent.

6.2 Predicting Performance

Performance of applications is predicted applying equation 2, using the *runtime_functions* from figure 3 and the average dependence constants in figure 5. For lack of space, although we have computed predicted runtimes for all configurations for each benchmark, we present numbers for only one benchmark, *fft*, in table 1. For now, consider only the first two columns of numbers in the table showing predicted and actual runtimes. Note that the prediction error is small. Though for *fft*, the error ranges from 0.13% to 11.68%, across all applications, the overall average error is small,

Configuration	Est. Cycle Count	Actual Cycle Count	Area Estimate	Feasible?
MAC, FPU	3.51156E+10	3.54307E+10	26%	Yes
MAC, Dual-ported Memory	1.29708E+11	1.29912E+11	16%	Yes
MAC, Pipelined Memory	1.27599E+11	1.27765E+11	13%	Yes
FPU, Dual-ported Memory	3.25175E+10	3.22097E+10	30%	Yes
FPU, Pipelined Memory	3.21419E+10	3.00624E+10	27%	Yes
Dual-ported Memory, Pipelined Memory	1.26009E+11	126691E+11	17%	Yes
MAC, FPU, Dual-ported Memory	3.06623E+10	2.79151E+10	36%	No
MAC, FPU, Pipelined Memory	3.06787E+10	2.79151E+10	33%	No
FPU, Dual-ported Memory, Pipelined Memory	3.13236E+10	2.89887E+10	37%	No
MAC, Dual-ported Memory, Pipelined Memory	1.2475E+11	1.26691E+10	23%	Yes
All four	3.01452E+10	2.68414E+10	43%	No
Best Estimated Feasible Config.: FPU, Pipelined Memory	3.21419E+10	3.00624E+10	27%	Yes
Best Actual Feasible Config.: FPU, Pipelined Memory	3.21419E+10	3.00624E+10	23%	Yes

Table 1: FFT benchmark: Evaluation of parameter combinations

ranging from 0.5% for combination (mac, dual-ported memory) to 5.16% for the combination (mac, FPU, pipelined memory).

6.3 Area Constrained Analysis

Our ultimate goal is to predict the configuration that achieves the best performance within a given area budget. Here we devise an experiment to predict the best configuration, and verify that it is best through exhaustive simulation. The experiment is as follows. The maximum allowable increase in area is fixed at 32%. The area increase from the use of MAC, hardware floating point, dual-ported memory and pipelined memory is assumed to be 6% [4], 20% [4], 10% and 7% respectively over the base architecture. For the fft application, table 1 shows the predicted and actual runtimes for the eleven possible combinations of parameters, along with their area estimates and feasibility.

From table 1, we see that the best feasible configuration for fft application is (FPU, pipelined memory) on the basis of predicted runtime values. This is verified to be true with actual runtime values, showing the predictive power of the method. Next we repeated the experiment in table 1 for the *spectral* application (not shown). We obtained the best predicted and actual feasible configuration to be (FPU, Dual-ported memory). This shows the value of our methodology as an application-specific methodology. It is able to judge that the best configuration is different for different benchmarks, based on compiler analysis.

7. CONCLUSIONS

This paper presents an automatic method to customize embedded application-specific instruction processors (ASIPs) based on compiler analysis. Present-day technologies for choosing parameter values rely on exhaustive simulation of the application set on all possible combinations of parameter values – a time-consuming and non-scalable procedure. We propose a compiler-based method that automatically derives the optimal values of parameters without simulating any configuration. Results show that for our benchmarks, the runtimes for different configurations are predicted with an average error of 2.5%. In the two area constrained customization problem we evaluate, our method is able to recommend the same configuration that is recommended by brute force exhaustive simulation.

8. REFERENCES

[1] *Technical Summary of the ARC Core.* ARC Cores Ltd, 2000. At http://www.arccores.com.

[2] P. Faraboschi, G. Brown, J. A. Fisher, G. Desoli, and F. Homewood. Lx: A Technology Platform for Customizable VLIW Embedded Processing. In *Proc. of the 27th Int'l Symp. on Computer Architecture (ISCA)*, Vancouver, British Columbia, Canada, June 2000.

[3] J. A. Fisher, P. Faraboschi, and G. Desoli. Custom-Fit Processors. Technical report, Hewlett Packard Laboratory, 1501, Page Mill Road, Palo Alto, CA 94304, 1997.

[4] M. Flynn. EE382 Processor Design: Silicon Area and Cost Models. *Course handout, EE382, Stanford Univ.*, 1999.

[5] N. Ghazal, R. Newton, and J. Rabaey. Predicting Performance Potential of Modern DSPs. In *Design Automation Conference*, Jun 2000.

[6] J. Gong, D. Gajski, and A. Nicolau. A Performance Evaluator for Parameterized ASIC Architectures. In *Proc. European Conf on Design Automation (EDAC)*, 1994.

[7] G. Goossens, J. Praet, D. Lanneer, W. Geurts, A. Kilfi, C. Liem, and P. Paulin. Embedded Software in Real-Time Signal Processing Systems: Design Technologies. *Invited paper, Proceedings of the IEEE*, 85(3), March 1997.

[8] T. V. K. Gupta, P. Sharma, M. Balakrishnan, and S. Malik. Processor Evaluation in an Embedded Systems Design Environment. In *Proceedings of the 13th International Conference on VLSI Design*, Jan 2000.

[9] O. Hebert, I. Kraljic, and Y. Savaria. A Method to Derive Application-Specific Embedded Processing Cores. In 8^{th} *Intl Wksp on Hardware/Software Codesign*. ACM, 2000.

[10] M. Kuulusa, J. Nurmi, J. Takala, P. Ojala, and H. Herranen. A Flexible Core for Embedded Systems. *IEEE Design and Test of Computers*, 13(4), October 1997.

[11] C. Lee and M. Stoodley. UTDSP BenchMark Suite. 1992. http://www.eecg.toronto.edu/~corinna/DSP/infrastructure-/UTDSP.html.

[12] S. Muchnick. *Advanced Compiler Design & Implementation.* Morgan Kaufmann, San Fran., CA, 1997.

[13] B. Shackleford, M. Yasuda, E. Okushi, H. Koizumi, H. Tomiyama, and H. Yasuura. Memory-CPU Size Optimization for Embedded Systems Designs. In *Design Automation Conference*. ACM, 1997.

[14] J. Turley. Tensilica CPU Bends to Designers' Will. *Microprocessor Report*, 13(3):12, March 8 1999.