Automatic Application-Specific Microarchitecture Reconfiguration

Shobana Padmanabhan, Ron K. Cytron, Roger D. Chamberlain, and John W. Lockwood

Dept. of Computer Science and Engineering
Washington University, St. Louis
shobana@arl.wustl.edu

Abstract

Applications for constrained embedded systems are subject to strict time constraints and restrictive resource utilization. With soft core processors, application developers can customize the processor for their application, constrained by resources but aimed at high application performance. With such freedom in the design space of the processor, however, comes complexity. We present here an automatic optimization technique that helps the developers with the processor microarchitecture customization.

A naive approach exploring all possible configurations is exponential with the number of parameters and hence is clearly infeasible, even with only tens of reconfigurable parameters. Instead, our approach runs in time that is linear with the number of parameter values, based on an assumption of parameter independence. This makes the approach feasible and scalable. For the dimensions that we customize, namely application runtime and hardware resources, we formulate their costs as a constrained binary integer nonlinear optimization program. Though the results are not guaranteed to be optimal, we find they are near-optimal in practice. Our technique itself is general and can be applied to other design-space exploration problems.

1. Introduction and Related Work

While application-specific, customized logic could dramatically improve the performance of an application, that approach is typically too expensive to justify its cost for most of the embedded applications. This has given rise to increased adoption of *soft core* processors, which are reconfigurable general purpose processors. Examples of soft core processors include Tensilica [26] with Stretch [24], MicroBlaze [29], and LEON [19].

There has been significant work centered around the idea of customizing a processor for a particular application or application set. Arnold and Corporaal [3] describe techniques for compilation given the availability of special function units. Atasu *et al.* [17] describe the design of instruction set extensions for flexible ISA systems. Choi *et al.* [8] examine a constrained application space in their instruction set extensions for DSP systems. Gschwind [11] uses both scientific computations as well as Prolog programs as targets for his instruction set extensions. Gupta *et al.* [12] developed a compiler that supports performance-model guided inclusion or exclusion of four functional units of multiple-accumulate, floating point, multiported memory and pipelined vs. non-pipelined memory unit. Systems that use exhaustive search for the exploration of the architecture parameter space are described in [13, 18, 22]. Heuristic design-space exploration for application-specific processors is considered in [9]. Pruning techniques are used to diminish the size of the necessary search space in order to find a Pareto-optimal design solution. In [5], the authors use a combination of analytic performance models and simulation-based performance models to guide the exploration of the design search space. Here, the specific application is in the area of sensor networks. Analytic models are used early, when a large design space is narrowed down to a "manageable set of good designs", and simulation-based models provide greater detail on the performance of specific candidate designs. The AutoTIE system [10] is a development tool from Tensilica that assists in the instruction set selection for Tensilica processors. This tool exploits profile data collected from executions of an application on the base instruction set to guide the inclusion or exclusion of candidate new instructions. [2] performs analytical (hierarchical) searching of parameters in their own dimensions, with some full parameter exploration to avoid local minimal, for tuning multi-level cache for low-energy embedded systems. [23] explores design options of instruction and data caches, branch predictor, and multiplier, by dividing the search space into piece-wise linear models and solving their results using integer linear programming.

Sponsored by NSF under grant CNS-0313203.

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

There are two main problems with most of these approaches. First, many of the approaches consider only a few parameters for customization or consider only a specific subsystem (such as cache) for a specific purpose (such as energy conservation). Such approaches do not scale well for the large number of reconfigurable parameters in a soft core processor. The second problem is the way application runtime is estimated using analytical models or measured using simulators, which are discussed subsequently.

Performance Measurement Analytic models can provide the quickest estimations of application performance, and such models are often derived directly from the source code. Examples of the use of analytic models include: [6], which describes an approach for the analytical modeling of runtime, idealized to the extent that cache behavior is not included; and [25], a classic paper on estimating software performance in a codesign environment, which reports accuracy of about $\pm 20\%$. However, for purposes of application performance improvement, $\pm 20\%$ is a wide deviation. These inaccurate predictions are due to the simplifying assumptions that are necessary to make analysis tractable and are notoriously common when analytic models are used. Moreover, application models often require sophisticated knowledge of the application itself. By contrast, simulation and the direct execution we use are both "black box" approaches that do not require knowledge of application implementation.

The method normally used to improve accuracy beyond modeling is *simulation*. Simulation toolsets commonly used in academia include: SimpleScalar [4], IMPACT [7], and SimOS [21]. Given the long runtimes associated with simulation modeling, it is a common practice to limit the simulation execution to only a single application run, not including the OS and its associated performance impact. SimOS does support modeling of the OS, but requires the simulation user manage the time/accuracy tradeoffs inherent in simulating such a large complex system.

We improve on this through the statistics module of Liquid architecture platform [15] which uses a hardware-based, non-intrusive profiler to count the number of clock cycles taken by an application executing directly on a soft core processor. Because it gives accurate runtime measures, we use this in our work.

2. Background

2.1. Liquid Architecture Platform

For our experiments, we use our *Liquid Architecture* platform [20]. Briefly, this platform instantiates a LEON2 [19] processor on a Xilinx VirtexE FPGA. It also provides a web interface to control the processor, run applications on it as well as profile runtime and microarchitecture parameters. This profiling is on application's direct execution on the processor and is hardware-based, non-intrusive and cycle-accurate.

Parameter	Values	Default
Instruction cache		
Sets	1-4	1
Set size	1,2,4,8,16,32,64KB 64KB requires 213 BRAM (i.e.) 33% more than available	4
Line size	4,8 words	8
Replacement	Random, LRR, LRU	Random
Data cache		
Sets	1-4	1
Set size	1,2,4,8,16,32,64KB 64KB requires 213 BRAM (i.e.) 33% more than available	4
Line size	4,8 KB	8
Replacement	Random, LRR, LRU	Random
Fast read	Enable/disable	Disable
Fast write	Enable/disable	Disable
Integer Unit		
Fast jump	Enable/disable	Enable
ICC hold	Enable/disable	Enable
Fast decode	Enable/disable	Enable
Load delay	1,2 clock cycles	1
Reg. windows	8, 16-32	8
Divider	radix2,none	radix2
Multiplier	none,iterative,m16x16, m16x16+pipelineRegs, m32x8,m32x16,m32x32,	m16x16
Synthesis options		
Infer Mult/Div	True/false	True

Figure 1. LEON reconfigurable parameters

2.2. LEON

LEON is an open source implementation of the SPARC V8 architecture, used by the European Space Agency. LEON's microarchitecture is parameterized along the systems of processor, bus, memory controller, peripherals, synthesis, clock, boot and debug. Processor comprises the subsystems of cache (separate instruction and data caches), Integer Unit (IU), Floating-point Unit, co-processor, Memory Management Unit (MMU) and Debug Support Unit (DSU). Boot and clock options are set once. We do not debug applications at runtime nor do our applications use peripherals or MMU. 64KB exceeds the available BRAM by 33%. FPU is excluded because two of the three supported interfaces (Sun's Meiko and Gaisler's GRFPU) are not free and the third one (LTH) is incomplete. Features such as SDRAM access, that are not part of out-of-the-box LEON distribution and require custom coding are also excluded. Figure 1 shows the parameters that impact the performance of our applications and their default values. Some are simple "enable or disable" while others take a range of values.

2.3. Constrained Binary Integer Nonlinear Programming

Linear Programming (LP) in *standard form* is the problem of minimizing a linear function, subject to a finite number of inequality constraints [16]. The following LP problem (or simply, linear program) has n variables, and k inequality constraints.

Minimize $Z = \mathbf{c}^T\mathbf{x}$

subject to

$$\mathbf{Ax} \geq \mathbf{b}$$
$$\mathbf{x} \geq \mathbf{0}$$

where $\mathbf{x} \in \mathbb{R}^n$ is the vector of decision variables to be solved for, $Z|\mathbb{R}^n -> \mathbb{R}$ is the objective function, $\mathbf{c} \in \mathbb{R}^n$ is the constants vector of cost coefficients, $\mathbf{A} \in \mathbb{R}^{k\times n}$ is the matrix of coefficients of k functional constraints, $\mathbf{b} \in \mathbb{R}^k$ is the constants vector of right-hand sides of functional constraints. *Other forms* of LP include maximization and inequalities that are $\leq$.

If the decision variables are restricted to be integers, it becomes Integer Linear Program (ILP). If they are further restricted to be binary-valued, the problem is Binary ILP. In addition, if the objective function or a constraint is nonlinear, then the problem is Binary Integer Nonlinear Program (BINLP).

ILP is exponential with the number of variables. Therefore, depending on how large the number of variables is and depending on whether or not there are special structures in the problem that some algorithms can exploit, some ILP problems may not be solved for optimal solution. With nonlinear optimization in the standard form, if the nonlinear objective function is not *convex* or if all the nonlinear constraints are not *concave* functions, the algorithm is *no* longer guaranteed to find the (global) minimum [14].

2.4. Cost Function

Chip Resource Cost Instantiating a soft core processor on FPGA utilizes resources and two fundamental ones are Lookup-tables (LUTs) and Block RAM (BRAM). Their utilizations are measured by actually *building* the processor, from it's source VHDL. The total LUTs and BRAM available on the Xilinx Virtex XCV2000E FPGA are 38,400 and 160 respectively and of them, the default (out-of-the-box) LEON configuration utilizes 14,992 (39%) and 82 (51%). Given the difference in their magnitudes, they are normalized as percentages and added together for a unified chip resource cost metric.

Application Runtime Cost Application runtime is measured by executing the application directly on the soft core processor (LEON) and counting the number of clock cycles the execution takes. We use the non-intrusive and cycle-accurate hardware-based profiler available through Liquid architecture platform.

Total Cost To be compatible with chip resource cost, application runtime cost is also normalized as a percentage and they are added together.

2.5. Benchmarks

The following applications are executed directly on LEON, without an operating system. Hence, they have been modified to avoid making system calls and using stdio. Liquid architecture platform now supports Linux and therefore, future work can make use of it.

Benchmark I - BLASTN Basic Local Alignment Search Tool (BLAST) [1] programs are the most widely employed set of software tools for comparing genetic material. BLASTN ("N" for *nucleotide*) is a variant of BLAST used to compare DNA sequences (lower-level than proteins) [20]. BLASTN is computation and memory-access intensive. It has approximately 163 lines of code and its runtime on the default LEON configuration is 10.6 seconds.

Benchmark II - CommBench DRR DRR is a Deficit Round Robin fair scheduling algorithm used for bandwidth scheduling on network links, as implemented in switches [28]. DRR is computation intensive. It has 117 lines of code and its runtime on the default LEON configuration is 5 minutes.

Benchmark III - CommBench FRAG Frag is an IP packet fragmentation application. IP packets are split into multiple fragments for which some header fields have to be adjusted and a header checksum computed, before being forwarded [28]. Frag is computation intensive. It has 150 lines of code and its runtime on the default LEON configuration is 2.5 minutes.

Benchmark IV - BYTE Arith Arith does simple arithmetics of addition, multiplication and division in a loop. It has been used to test processor speed for arithmetic. Arith is not memory intensive. It has 77 lines of code and its runtime on the default LEON configuration is 32 seconds.

3. Approach

Our goal is to improve performance of a given application through automatic reconfiguration of processor microarchitecture to meet the application's requirements and constraints closely. The approach we take is to consider *all*

parameters that have a bearing on application runtime or hardware resources and to use *actual* measurements (costs) rather than estimates, to obtain more *accurate* customization. Despite these, we want our optimization technique to be *feasible and scalable*.

The challenge from considering all parameters is that, it makes the search space huge. The 79 parameter values in Figure 1, which is really a subset of the reconfigurable parameters in LEON, results in $3,641,573,376$ exhaustive configurations. The second challenge comes from measuring *actual* costs. Costs are measured for all the dimensions that are being optimized and/ constrained. Currently, we optimize application runtime and FPGA resources. The execution times for our benchmarks range from 16 seconds to 9 minutes. We leave it for future work to address very long execution times, possibly through a smart sampling technique. Hardware resource utilizations are measured by actually building processor configurations from the source VHDL. Each build is very time-consuming, on the order of 30 minutes, even on modern computers. These two challenging makes the customization harder than a traditional optimization problem because they make it infeasible to do exhaustive enumeration to build an exact model and search for the best solution.

The next best approach is to build an approximate model and solve for an exact solution. We build the model by assuming *parameter independence* and restricting each parameter to it's own dimension. Though our results are no longer guaranteed to be optimal in all cases, Section 5 demonstrates that they are near-optimal in practice. With the assumption of parameter independence, the number of configurations is *linear* in the number of parameter values, 52 for the parameters in Figure 1. Even if the remaining parameters benefit other applications, it would still be only 100 configurations, which is still feasible and scalable.

We solve for optimal solution by formulating the model as a constrained Binary Integer Nonlinear Problem. Although the search space is built by considering parameters in their own dimensions, the optimization algorithm evaluates points in between. These points represent configurations that have more than one parameter changed simultaneously. The solver assigns costs for these points through an *approximation* of actual costs provided by us in the model.

The approach to building the model is summarized as follows. We begin with the default LEON configuration that comes out-of-the-box. We call this the *base* configuration. We then perturb one parameter at a time, build the processor configuration and measure it's chip cost. Thirdly, we execute the application on each configuration and measure the runtime. Finally, we formulate these costs into a BINLP problem and solve for optimal solution, using the commercial solver of Tomlab Mixed Integer Nonlinear Programming solver [27]. Tomlab is a plug-in to Matlab and solves our formulation in *seconds*. The solution obtained is the recommended microarchitecture configuration for the given application.

4. Problem Formulation

We formulate the problem of automatic application-specific customization of soft core processor microarchitecture as a Binary Integer Nonlinear problem (BINLP). The objective is to meet the applications runtime requirements and FPGA resource restrictions. The constraint is to select a *valid* microarchitecture configuration that *fits* in the available chip resources.

We begin the reconfiguration with the default (out-of-the-box) LEON configuration that we call as the *base configuration*. The %LUTs and %BRAM remaining unutilized after the base configuration are denoted by L and B respectively. From the base configuration, we change the parameter values one at a time, build a new processor configuration x_i and execute the application on it. For each x_i, the difference (in percentage) in LUTs, BRAM and application runtime over the base configuration are denoted by λ_i, β_i and ρ_i respectively.

4.1. Objective Function

The objective function is to minimize the costs of the dimensions being optimized. The dimensions that we optimize are application runtime and chip resources and the following equation minimizes their costs. We use *weights* to optimize certain dimensions over others.

$$\text{Minimize} \sum_{i=1}^{n=52} [w_1(\rho_i x_i) + w_2((\lambda_i + \beta_i)x_i)]$$

w_1 and w_2 are independent. w_1 is made to dominate w_2 for **application runtime optimization** and w_2 is made to dominate for **FPGA resource optimization**.

4.2. Constraints

Parameter Validity Constraints x_i represents a new processor configuration resulting from a change in *one parameter value* from the *base* configuration. x_i is binary (i.e.) it represents two values—*on/ off* or *two integer values*. That implies that for parameters with more than two values, more than one x_i will be used. Therefore, for such parameters, we need to ascertain that only one variable is selected. All such constraints are presented below.

$\Sigma_{i=1}^{3} x_i \leq 1$ (icache nsets)
$\Sigma_{i=4}^{8} x_i \leq 1$ (icache setsize)
$\Sigma_{i=10}^{11} x_i \leq 1$ (icache replacement policy)
$\Sigma_{i=12}^{14} x_i \leq 1$ (dcache number of sets)
$\Sigma_{i=15}^{19} x_i \leq 1$ (dcache setsize)

$\Sigma_{i=21}^{22} x_i \leq 1$ (dcache replacement policy)
$\Sigma_{i=30}^{46} x_i \leq 1$ (IU nwindows)
$\Sigma_{i=47}^{51} x_i \leq 1$ (different hardware multipliers)
The binary variables not constrained here are as follows. x_9 represents icache linesize, x_{20} dcache linesize, x_{23} Integer Unit (IU) fast *jump*, x_{24} Integer Condition-Code, x_{25} fast instruction-decode, x_{26} load-delay, x_{27} dcahe fast read, x_{28} hardware divider, x_{29} infer-multiplier and x_{52} dcahe fast write.

There are additional constraints imposed by LEON. The icache and dcache replacement policy of LRR (Least Recently Replaced) can be used only with 2-way associativity (2 sets) and LRU (LR Used) with all multi-way associativity.

$x_{10} - x_1 \leq 0$
$\Sigma_{i=1}^{3} x_i - x_{11} \geq 0$
$x_{21} - x_{12} \leq 0$
$\Sigma_{i=12}^{14} x_i - x_{22} \geq 0$

FPGA Resource Constraints For the FPGA resources considered, their utilization for each x_i should fit in what is available after the base configuration.
$\Sigma_{i=1}^{52} \lambda_i x_i \leq L$ and
$\Sigma_{i=1}^{52} \beta_i x_i \leq B$

Cache size (of both icache and dcache) is expressed in terms of two parameters in LEON viz. number of cache sets and size of each set. Accounting for this, the constraint equations for hardware resources become:

$$(1 + x_1 + 2x_2 + 3x_3) \times (\Sigma_{i=4}^{8} \lambda_i x_i) + (1 + x_{12} + 2x_{13} + 3x_{14}) \times (\Sigma_{i=15}^{19} \lambda_i x_i) + \Sigma_{i=1}^{3} \lambda_i x_i + \Sigma_{i=9}^{11} \lambda_i x_i + \Sigma_{i=12}^{14} \lambda_i x_i + \Sigma_{i=20}^{52} \lambda_i x_i \leq L$$

$$(1 + x_1 + 2x_2 + 3x_3) \times (\Sigma_{i=4}^{8} \beta_i x_i) + (1 + x_{12} + 2x_{13} + 3x_{14}) \times (\Sigma_{i=15}^{19} \beta_i x_i) + \Sigma_{i=1}^{3} \beta_i x_i + \Sigma_{i=9}^{11} \beta_i x_i + \Sigma_{i=12}^{14} \beta_i x_i + \Sigma_{i=20}^{52} \beta_i x_i \leq B$$

The *convexity* of these *nonlinear* functions is conditional on the values of x_i. That means, the optimization algorithm is no longer guaranteed to find global optimum in *all* cases. Therefore, to optimize the problem formulation, we leave the constraint on LUTs as a linear function, since variation in LUTs utilization is very minimal. We analyze the effect of this in Section 6.

5. Analysis

In this section, we analyze the impact of our assumption of parameter independence. The naive approach of comparing our solution to the one obtained by generating all configurations exhaustively is infeasible for us. The next logical approach is to scale down the problem space such that it becomes feasible to generate all configurations exhaustively. When these two solutions compare favorably, we show that our optimization algorithm works as well as can be expected.

BLASTN: exhaustive: dcache sets,setsize				
nsets	Setsz(KB)	Runtime(sec)	LUTs(%)	BRAM(%)
1	1	10.71	38	47
1	2	10.64	38	48
1	4	10.60	39	51
1	8	10.54	39	56
1	16	10.50	38	68
1	32	10.22	38	90
2	1	10.58	39	49
2	2	10.55	39	51
2	4	10.53	39	56
2	8	10.50	39	68
2	16	10.22	39	90
3	1	10.56	39	51
3	2	10.54	39	55
3	4	10.51	39	62
3	8	10.45	39	79
4	1	10.55	39	53
4	2	10.53	39	58
4	4	10.50	39	68
4	8	10.22	39	90
Optimal runtime				
2	16	10.22	39	90

Figure 2. Dcache exhaustive for BLASTN

We chose the subsystem of dcache for this purpose because we had manually optimized the cache subsystem for BLASTN application in [20]. More fundamentally, cache subsystem has tangible variations in application performance and chip resource utilization, for changes in parameter values. As enumerated in Section 4, dcache has 7 reconfigurable parameters of number of sets, size of each set, associativity, line size, replacement policy, fast read and write options. The number of integer values of these parameters are 4, 7, 4, 2, 3, 2 and 2 respectively. Their exhaustive combinations are 2,688 and it would take at least 56 days to generate them all. That is not scalable and therefore, we consider only two parameters–number of sets and set size, which result in 28 combinations. We chose these two parameters because perturbing these affects both LUTs and BRAM utilization, at varying degrees. The base configuration has 1 set of 4KB size.

BLASTN Figure 2 shows BLASTN runtimes and chip resource costs for the *exhaustive* combinations of dcache sets and set size. Optimizing for runtime, a simple sort yields the optimal configuration of 2 sets of 16KB each (i.e.) a total of 32KB. The performance gain is 3.63% over the base configuration, utilizing no additional LUTs but 39% more BRAM than the base configuration.

We then compare this to the configurations that we evaluate as per our approach, the optimizer. Figure 3 shows this. Optimizing only for application runtime, the configuration we select is set size 32KB, which is the same cache size as selected by the exhaustive search but organized slightly

Base configuration				
1	4	10.60	39	51
BLASTN: optimizer: dcache sets,setsize ($w_1 = 100, w_2 = 0$)				
Sets	Setsz(KB)	Runtime(sec)	LUTs(%)	BRAM(%)
2	4	10.53	39	56
3	4	10.51	39	62
4	4	10.50	39	68
1	1	10.71	38	47
1	2	10.64	38	48
1	4	10.60	39	51
1	8	10.54	39	56
1	16	10.50	38	68
1	32	10.22	38	90
Dcache optimization for BLASTN runtime				
1	32	10.22	38	90

Figure 3. Dcache optimization for BLASTN runtime

Optimizer: dcache sets,setsize ($w_1 = 100, w_2 = 0$)					
	Sets	Setsz(KB)	Time(sec)	LUT%	BRAM%
CommBench DRR					
Exhaust	1	32	261.609	38	90
	2	16	261.609	39	90
Optimiz	2	16	261.609	39	90
CommBench FRAG					
Exhaust	1	32	147.869	38	90
Optimiz	2	16	147.869	39	90
BYTE Arith					
Exhaust	No effect, as application is not data intensive				
Optimiz	No effect, as application is not data intensive				

Figure 4. Dcache optimization for DRR, FRAG, Arith

differently. The performance gain with this configuration is 3.61%, which is 0.02% less than the optimal configuration from the exhaustive approach; LUTs utilization is 1% less here and BRAM is the same.

The fact that our optimization was able to achieve performance gain within 0.02% difference from the exhaustive solution and with 1% reduction in LUTs (chip resource cost), in spite of the assumption of parameter independence, is very encouraging.

Other Benchmarks Results for the other benchmarks discussed in Section 2.5 are even better, as they match the solution from exhaustive approach. They are shown in Figure 4. This gives us further confidence that our customization finds valid and near-optimal configurations, despite the assumption of parameter independence.

Further Observations The results for DRR and FRAG are 2x16 configurations. These are *not* configurations that we provide directly in the model. This demonstrates that, while we construct the search space reconfiguring parameters in their own dimensions, the optimization algorithm considers points in between, which are points reconfigured simultaneously in many dimensions. Next, the fact that we are able to build the solutions proves that we generate *valid* configurations. Finally, the configurations selected are indeed application-specific.

6. Results

The research objectives of our experiments are to find out how much improvement we gain from the application-specific microarchitecture customization and to demonstrate that the customization is indeed application-specific. The results in Section 5 addresses both, for a subset of dcahe parameters. This section presents results for *all* LEON parameters shown in Figure 1. Section 6.1 shows the performance gains to be 6.15%–19.39% and Figure 5 and Figure 7 show that the customization is indeed application-specific. Due to space constraints, we are restricted here to showing chip resource and application runtime costs only for our *solutions*, rather than for all the configurations that we consider.

We first present results of optimizing runtime over chip resources, by setting w_1 to be much higher than w_2, and then, vice versa. Figures show only the parameters that are reconfigured from the base configuration.

6.1. Application Performance Optimization

We optimize application performance over chip resources by setting $w_1 = 100$ and $w_2 = 1$. Figure 5 shows the parameters reconfigured from the base configuration, along with results from the actual build of the solution. Based on the latter, runtime decrease for the four applications of BLASTN, DRR, FRAG and Arith are 11.59%, 19.39%, 6.15% and 6.49%, over the runtimes on their respective base configurations. The linear approximations performed by the optimizer estimate the performance improvements to be 11.77%, 39.14%, 7.67% and 6.49%, respectively. The range of overestimation is 0–19.75%. Due to space constraints, we present costs only for BLASTN reconfigurations, in Figure 6.

The performance gains come at the expense of additional chip resources. The increase in chip resource utilization, expressed as a tuple of LUTs and BRAM, are $(0\%, 39\%)$, $(0\%, 39\%)$, $(8\%, 42\%)$ and $(1\%, -3\%)$ respectively. The approximations performed by the optimizer are $(-4\%, 36\%)$, $(-4\%, 41\%)$, $(-4\%, 44\%)$ and $(-2\%, -4\%)$, respectively. We consistently underestimate LUTs utilization; our estimates for BRAM are mixed, from -2% to 3%.

Cost Approximations As we saw in Section 4, we simplified the cost function for LUTs to be linear while leaving it nonlinear for BRAM. To evaluate the simplification, we present what the nonlinear approximations would be for LUTs in Figure 5. As seen there, our underestimations

Application runtime optimization ($w_1 = 100, w_2 = 1$)					
Param	Base	BLAST	DRR	FRAG	Arith
icachsetsz	4	2	2	4	4
icachlinesz	8	4	4	4	4
dcachsets	1	1	2	2	1
dcachsetsz	4	32	16	16	1
dcachlinesz	8	4	4	4	8
dcachreplace	rnd	LRU	LRR	LRU	rnd
fastjump	on	off	off	off	off
icchold	on	off	off	off	off
divider	radix2	none	none	none	radix2
multiplier	16x16	32x32	32x32	32x32	32x32
Base configuration					
runtime(sec)	N/A	10.60	297.98	150.75	32.33
Cost approximations by the optimizer					
runtime(sec)	N/A	9.35	181.35	139.20	30.23
LUTs%	39	35	35	35	37
LUTs%-nonlin	*39*	*35*	*34*	*34*	*37*
BRAM%	51	87	92	95	47
BRAM%-lin	*51*	*87*	*75*	*78*	*47*
Actual synthesis					
runtime(sec)	N/A	9.37	240.20	141.48	30.23
LUTs%	39	39	39	47	40
BRAM%	51	90	90	93	48

Figure 5. Application runtime optimization

Param	Runtime(sec)	LUTs(%)	BRAM(%)
icachesetsz2	10.60	39	48
icachelinesz4	10.60	38	51
dcachesetsz32	10.22	38	90
dcachelinesz4	10.58	39	51
nofastjump	10.60	38	51
noicchold	10.24	39	51
nodivider	10.60	37	51
multiplierm32x32	10.12	40	51

Figure 6. BLASTN runtime optimization costs

would be slightly higher and hence, worse. In addition, to demonstrate how *better* the nonlinear cost function *is* over the linear for BRAM, we present the linear approximations also. Space constrains restrict similar analysis in Section 5.

Comparison with Dcache Optimization Given our assumption of parameter independence, an interesting observation is to compare the customization in dcache here to the one from optimizing only dcache in Section 5. However, the weights in the objective function are slightly different– for the former, $w_1 = 100$ and $w_2 = 1$ and for the latter $w_1 = 100$ and $w_2 = 0$. The resulting dcache configurations are identical for all applications except Arith. For Arith, it was 1x4 in Section 5 but here it is 1x1. This is because of the chip resource consideration resulting from $w_2 = 1$.

6.2. FPGA Resource Optimization

We optimize chip resources over application performance by setting $w_1 = 1$ and $w_2 = 100$. Figure 7 shows the parameters reconfigured from the base configuration,

Chip resource optimization ($w_1 = 1, w_2 = 100$; *=sub-optimal)					
Param	Base	BLAST*	DRR*	FRAG	Arith*
icachsetsz	4	2	2	4	2
icachlinesz	8	4	4	4	4
dcachsets	1	1	1	1	1
dcachsetsz	4	2	2	1	2
dcachlinesz	8	4	4	4	8
dcachreplace	rnd	rnd	rnd	rnd	rnd
fastjump	on	off	off	off	off
icchold	on	off	off	off	off
divider	radix2	none	none	none	radix2
registers	8	28*	31*	8	30*
multiplier	16x16	iter	iter	iter	iter
Cost approximations by the optimizer					
runtime(sec)	N/A	13.86	355.82	153.19	44.08
LUTs%	39	34	32	32	34
LUTs%-nonlin	*39*	*34*	*32*	*32*	*34*
BRAM%	51	47	47	47	47
BRAM%-lin	*51*	*47*	*47*	*47*	*47*
Actual synthesis					
runtime(sec)	N/A	13.85	347.91	151.40	44.08
LUTs%	39	37	37	36	38
BRAM%	51	48	48	48	48

Figure 7. Chip resource optimization

along with results from the actual build of the solution. Based on the latter, decrease in chip resource utilization are $(2\%, 3\%)$, $(2\%, 3\%)$, $(3\%, 3\%)$ and $(1\%, 3\%)$. The approximations performed by our optimization algorithm estimate the chip resource savings to be $(5\%, 4\%)$, $(7\%, 4\%)$, $(7\%, 4\%)$ and $(5\%, 4\%)$. We consistently overestimate the chip savings; for LUTs, the range is 3—5% and for BRAM, it is always 1%.

Similar to application runtime optimization, here also, we present the nonlinear approximations for LUTs and linear approximations for BRAM in Figure 7.

The savings in chip resources come at a loss of application performance, often significant – 30.66% for BLASTN, 16.76% for DRR, 0.43% for FRAG and 36.34% for Arith.

7. Conclusion and Future Work

We have presented a heuristic for automatic application-specific reconfiguration of a soft core processor microarchitecture. This approach is linear in the number of reconfigurable parameters, with an assumption of parameter independence, to make the approach feasible and scalable. The performance gains over the *base* configuration are near-optimal in practice, despite our simplifying assumption. More importantly, our technique empowers application developers to do performance-resource tradeoffs in hours and without detailed knowledge of the architecture.

Future work can recast our nonlinear constraints so that they are convex functions for all values of x_i. This will guarantee that the optimization algorithm finds the global optimum. We can also analyze the cost approximations performed by the optimization algorithm and explore more so-

phisticated approximations. As extensions to our model, we can include power and energy optimizations, runtime sampling to facilitate analysis of long-running applications, running applications on an operating system and supporting ISA level customization. By integrating our solution with open source soft core processors, we can contribute back to the community. Finally, and more interestingly, we can evaluate our technique on other configuration and feature management problems.

Acknowledgements

We thank the referees for their comments, which have improved this paper.

References

[1] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, et al. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–10, 1990.

[2] F. V. Ann Gordon-Ross, Chuanjun Zhang and N. Dutt. *Tuning caches to applications for low-energy embedded systems*. Kluwer Academic Pub, June 2004.

[3] M. Arnold and H. Corporaal. Designing domain-specific processors. In *Proc. of the 9th Int'l Symp. on Hardware/Software Codesign*, pages 61–66, Apr. 2001.

[4] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, Feb. 2002.

[5] A. Bakshi, J. Ou, and V. K. Prasanna. Towards automatic synthesis of a class of application-specific sensor networks. In *Proc. of Int'l Conf. on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 50–58, 2002.

[6] C. Brandolese, W. Fornaciari, F. Salice, and D. Sciuto. Source-level execution time estimation of C programs. In *Proc. of the 9th Int'l Symp. on Hardware/Software Codesign*, pages 98–103, Apr. 2001.

[7] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu. IMPACT: an architectural framework for multiple-instruction-issue processors. In *Proc. of the 18th Int'l Symp. on Computer Architecture*, May 1991.

[8] H. Choi, J.-S. Kim, C.-W. Yoon, I.-C. Park, S. H. Hwang, and C.-M. Kyung. Synthesis of application specific instructions for embedded DSP software. *IEEE Trans. on Computers*, 48(6):603–614, June 1999.

[9] D. Fischer, J. Teich, M. Thies, and R. Weper. Efficient architecture/compiler co-exploration for ASIPs. In *Proc. of Int'l Conf. on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 27–34, 2002.

[10] D. Goodwin and D. Petkov. Automatic generation of application specific processors. In *Proc. of Int'l Conf. on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 137–147, 2003.

[11] M. Gschwind. Instruction set selection for ASIP design. In *Proc. of the 7th Int'l Symp. on Hardware/Software Codesign*, pages 7–11, May 1999.

[12] T. V. K. Gupta, R. E. Ko, and R. Barua. Compiler-directed customization of ASIP cores. In *Proc. of the 10th Int'l Symp. on Hardware/Software Codesign*, pages 97–102, May 2002.

[13] O. Hebert, I. C. Kraljic, and Y. Savaria. A method to derive application-specific embedded processing cores. In *Proc. of the 8th Int'l Symp. on Hardware/Software Codesign*, pages 88–92, May 2000.

[14] F. Hillier and G. Lieberman. *Introduction to Operations Research*. Tata McGraw-Hill, Seventh reprint edition, 2004.

[15] R. Hough, P. Jones, S. Friedman, R. Chamberlain, J. Fritts, J. Lockwood, and R. Cytron. Cycle-accurate microarchitecture performance evaluation. In *Proc. of Workshop on Introspective Architecture*, Feb. 2006.

[16] H. Karloff. *Linear Programming*. Birkhauser Boston, August 1991.

[17] P. I. Kubilay Atasu, Laura Pozzi. Automatic application-specific instruction-set extensions under microarchitectural constraints. In *Proc. of Design Automation Conf.*, June 2003.

[18] M. Kuulusa, J. Nurmi, J. Takala, P. Ojala, and H. Herranen. A flexible DSP core for embedded systems. *IEEE Design and Test of Computers*, 14(4):60–68, 1997.

[19] LEON Specification. `http://www.gaisler.com-/doc/leon2-1.0.21-xst.pdf`, 2003.

[20] S. Padmanabhan, P. Jones, D. V. Schuehler, S. J. Friedman, P. Krishnamurthy, H. Zhang, R. Chamberlain, R. K. Cytron, J. Fritts, and J. W. Lockwood. Extracting and improving microarchitecture performance on reconfigurable architectures. *International Journal of Parallel Programming*, 33(2–3):115–136, June 2005.

[21] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod. Using the SimOS machine simulator to study complex computer systems. *ACM Trans. on Modeling and Computer Simulation*, 7(1):78–103, Jan. 1997.

[22] B. Shackleford, M. Yasuda, E. Okushi, H. Koizumi, H. Tomiyama, and H. Yasuura. Memory-CPU size optimization for embedded system designs. In *Proc. of Design Automation Conf.*, pages 246–251, June 1997.

[23] T. Sherwood, M. Oskin, and B. Calder. Balancing design options with sherpa. In *CASES '04: Proceedings of the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 57–68, New York, NY, USA, 2004. ACM Press.

[24] Stretch, Inc. S5500 Software-Configurable Processor. `http://www.stretchinc.com`.

[25] K. Suzuki and A. Sangiovanni-Vincentelli. Efficient software performance estimation methods for hardware/software codesign. In *Proc. of Design Automation Conf.*, pages 605–610, June 1996.

[26] Tensilica, Inc. `http://www.tensilica.com`.

[27] Tomlab Optimization. Users Guide for TOMLAB /MINLP. `http://www.tomlab/.biz/docs/-TOMLAB_MINLP.pdf`.

[28] T. Wolf and M. A. Franklin. Commbench - a telecommunications benchmark for network processors. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 154–162, Austin, TX, Apr. 2000.

[29] Xilinx Corp. MicroBlaze Product Brief, FAQ. `http://www.xilinx.com/ipcenter/-processor_central/microblaze.htm`, May 2001.