SPECIAL ISSUE PAPER

Finding near-perfect parameters for hardware and code optimizations with automatic multi-objective design space explorations

Ralf Jahr 1,*,†, Horia Calborean 2, Lucian Vintan 2 and Theo Ungerer 1

1 Department of Computer Science, University of Augsburg, 85135 Augsburg, Germany
2 “Lucian Blaga” University of Sibiu, Computer Science & Engineering Department, E. Cioran Str., No. 4, Sibiu - 550025, Romania

SUMMARY

In the design process of computer systems or processor architectures, typically many different parameters are exposed to configure, tune, and optimize every component of a system. For evaluations and before production, it is desirable to know the best setting for all parameters. Processing speed is no longer the only objective that needs to be optimized; power consumption, area, and so on have become very important. Thus, the best configurations have to be found in respect to multiple objectives. In this article, we use a multi-objective design space exploration tool called Framework for Automatic Design Space Exploration (FADSE) to automatically find near-optimal configurations in the vast design space of a processor architecture together with a tool for code optimizations and hence evaluate both automatically. As example, we use the Grid ALU Processor (GAP) and its postlink optimizer called GAPtimize, which can apply feedback-directed and platform-specific code optimizations. Our results show that FADSE is able to cope with both design spaces. Less than 25% of the maximal reasonable hardware effort for the scalable elements of the GAP is enough to achieve the processor’s performance maximum. With a performance reduction tolerance of 10%, the necessary hardware complexity can be further reduced by about two-thirds. The found high-quality configurations are analyzed, exhibiting strong relationships between the parameters of the GAP, the distribution of complexity, and the total performance. These performance numbers can be improved by applying code optimizations concurrently to optimizing the hardware parameters. FADSE can find near-optimal configurations by effectively combining and selecting parameters for hardware and code optimizations in a short time. The maximum observed speedup is 15%. With the use of code optimizations, the maximum possible reduction of the hardware resources, while sustaining the same performance level, is 50%. Copyright © 2012 John Wiley & Sons, Ltd.

Received 20 December 2011; Revised 10 October 2012; Accepted 5 November 2012

KEY WORDS: automatic design space exploration; multi-objective optimization; hardware complexity estimation; code optimization

1. INTRODUCTION

As in the last decade, the complexity of novel processor architectures is going to increase in the future. This is caused by the steadily growing number of transistors. Although for early processor architectures the available number of transistors was a limiting bound, the architectures proposed more recently have the freedom to use lots of hardware. To cope with the increasing complexity of designs, while preserving effective usage of hardware resources, is one of the challenges to be solved by new approaches.

*Correspondence to: Ralf Jahr, Department of Computer Science, University of Augsburg, 85135 Augsburg, Germany.
†E-mail: jahr@informatik.uni-augsburg.de

Copyright © 2012 John Wiley & Sons, Ltd.
The most common approaches are increasing the cache sizes and replicating processor cores, which leads to multicore and many-core designs. Alternatively, novel processor architectures such as TRIPS (with EDGE [1]), RAW [2], or the Grid ALU Processor (GAP) can be used. The GAP combines elements of superscalar processor architectures with a coarse-grained reconfigurable array of functional units (FUs). Its goal is to speed up the execution of sequential instructions streams.

All novel processor architectures expose many parameters to allow tuning, configuration, and optimization. Examples for parameters are the number of processor cores, cache sizes, or memory bandwidth. By changing them, every component of the processor can be configured, tuned, and optimized. These parameters form a huge design space. With multiple objectives and under specific constraints, as for example timing behavior or the availability and affordability of hardware resources, good combinations of parameters have to be found. Knowing the best possible configurations is especially important for benchmarking of processor architectures and before tape-out, so only the best-known processor configurations should actually be selected for production.

It is very hard for system designers to cope with this increased complexity. Tools for automatic design space exploration (ADSE) are very convenient for this job. One of them is the Framework for ADSE (FADSE, [3, 4]), which has its focus on processor architectures.

FADSE has been developed to intelligently explore relevant subspaces of a huge design space using state-of-the-art evolutionary and bio-inspired search algorithms. Therefore, it evaluates many different configurations with a heuristic algorithm. Each configuration is formed by a set of parameters, where one value is selected for each parameter. The main goal of the design space exploration (DSE) is to find a very good approximation of the Pareto front consisting of the best-known configurations regarding multiple objectives.

As processor architectures become more complex and diverse, the optimization of programs by compilers and code optimizers in general also becomes harder because the quality of the settings for these programs can be very different for apparently similar target platforms. Therefore, we propose an ADSE to solve this challenge, too. For code optimizations, we use GAPtimize, a postlink optimizer to apply feedback-directed platform-specific optimizations on binaries for the execution on the GAP.

The main contributions of this paper, which is an extended version of [5], are as follows: (i) an improved version of FADSE with higher robustness and a much higher degree of parallelism; (ii) the introduction of a model to estimate the hardware complexity of different configurations of the GAP; (iii) a description of the performance achievable with the GAP on configurations with varying complexities; (iv) a guideline on how to spread complexity on the different components of the GAP and how to chose its hardware parameters for configurations of high quality; and (v) finding that FADSE can also be used to find good parameters for software optimizations. The article also gives an example on how to interpret the results of a DSE in a sophisticated manner by not only describing, for example, the best possible performance but also showing how to choose parameters to reach it.

Details on FADSE, GAP, GAPtimize, and the objectives used in the DSE are presented in Section 2. The results of the DSE are explained in Section 3. The presented work is put in context with related work in Section 4. Section 5 concludes the paper.

2. RESEARCH METHODOLOGY

In the following section, we introduce the components of our case study and provide the basis for the DSE by presenting parameters and objectives.

2.1. Basic concepts

The purpose of a multi-objective DSE of computer architectures is to find good configurations, in the parameter space (Figure 1), by taking into consideration multiple objectives (speed, hardware complexity, etc.), resulting in a point in the objective space.
AUTOMATIC OPTIMIZATION OF PARAMETERS FOR HARDWARE AND SOFTWARE

Figure 1. Mapping of the points from parameter space (left) to objective space (right) by evaluating them; the example shows a problem with two objectives and two parameters, both objectives should be minimized.

Pareto-efficiency-based heuristic algorithms can be used for this purpose. When multiple (conflicting) objectives are optimized, a great probability that not a single best configuration (individual) exists. One configuration might be better on one objective but worse on another one. There are situations when an order cannot be established between individuals. If an individual is worse on one objective and equal or worse on all the other objectives, compared with another individual, then this individual is dominated by the better one. If one individual is better on one objective but worse on another then the two individuals are called nondominated between themselves. All the individuals found by the heuristic algorithm from a population that are not dominated by any other individual in the same population form, in the objective space, the Pareto front approximation. If the individuals are optimal, they form the true Pareto front (unknown in complex real-life problems).

2.2. Framework for automatic design space exploration

FADSE allows us to perform ADSEs using the state-of-the-art evolutionary multi-objective algorithms implemented in jMetal [6] (nondominated sorting genetic algorithm (NSGA-II) [7], SPEA2 [8], speed-constrained multi-objective particle swarm optimization (SMPSO) [9], and many others). The main characteristics of FADSE were already presented [10, 11]; in the scope of this work, FADSE has been improved fundamentally to reduce the time needed for a DSE process and to increase its robustness.

The algorithms used for the DSE have been modified to be run in a distributed manner [4]. Creating new individuals for a new generation is decoupled from evaluating individuals, which allows running the simulations necessary for the evaluation of individuals in parallel (down to the core level). As result, the sequential part is only the central part of the heuristic algorithm, which computes a generation in a matter of milliseconds. This is insignificant compared with the minutes or hours needed to evaluate a single individual.

FADSE works on personal computers with commodity networks as well as on supercomputers (it was for example tested on an IBM HPC with 120 Intel Xeon cores running RedHat Linux).

After a DSE is run, it is important to understand the quality of the generated results. For this task, FADSE now offers metrics that do not require the true Pareto front. It supports the calculation of the hypervolume, which is the volume enclosed between the Pareto front approximation and the axes in a maximization problem, coverage, which is the fraction of individuals from one population dominated by the individuals from another population, the seven-point average distance [12] and the hypervolume two-set difference [13]. These metrics can be used to do the following: (i) compare different DSEs and (ii) measure the progress of a certain DSE algorithm.
FADSE also integrates various improvements to increase robustness, which take into consideration situations when a simulator or a computer crashes or a connection is interrupted, allowing FADSE to continue working without affecting the results.

The framework was designed so that it can run almost any existing simulator, which can be configured with a set of parameters by writing a specific connector. The configuration of FADSE is stored in an XML file, in which the user can set the simulator parameters, the simulated architecture, and a set of constraints (rules). These constraints reduce the search space and thus help the algorithm to find a quasi-optimal solution faster [4].

For a further speedup of explorations, the improved version of FADSE can use a database to store the results of the evaluations of individuals. This allows the reuse of already calculated results, which considerably reduces the time required to find a good solution.

Recent improvements to FADSE, which is available as open source‡, allow the designer to express his or her knowledge in a human readable form using fuzzy rules [14, 15].

2.3. Grid ALU Processor

The GAP is a processor architecture to speed up the execution of single-threaded instruction streams. It comprises a superscalar-like in-order front end consisting of a fetch unit and a decode unit that are used together with a novel configuration unit (Figure 2). This unit is able to concurrently map independent instructions as well as data or control flow-dependent instructions dynamically onto an array of FUs, a branch control unit, and several load/store units to handle memory accesses.

The array of FUs is organized in columns and rows. Each column is assigned dynamically and per configuration to one architectural register. Every instruction is mapped to the column whose register matches the instruction’s output register. The rows of the array are used to model dependencies between instructions. A bimodal branch predictor effectively maps control dependencies onto the array.

So that configurations can be saved for repeated execution, all elements of the array are equipped with memories that form configuration layers working similar to a trace cache [16]. Typically, 2,

‡Home page of FADSE: http://code.google.com/p/fadse/
4, 8, 16, 32, or 64 configuration layers are available; the number of layers is an example for a dimension of the design space.

A cycle-accurate and signal-accurate simulator has been developed to evaluate the architecture. It uses the portable instruction set architecture derived from a MIPS instruction set architecture; hence, the simulator can execute the identical program files as the SimpleScalar simulation tool set [17]. Detailed information about the processor is given by Uhrig et al. [18] and Shehan et al. [19].

Unlike traditional architectures, the GAP has been designed to be scalable, that is, to effectively use smaller or larger die areas. This is mainly achieved by the different sizes of the array of FUs (from 16 to 992 FUs) as well as size and organization of the caches.

The configurability of the architecture together with the ability to execute legacy code without any modification on any configuration of the GAP makes it very interesting as a candidate for a case study with FADSE. A command-line interface has been built for the GAP simulator to invoke it with FADSE. To be able to detect unexpected behavior of the simulator and hence to ensure robustness, FADSE observes an often-updated log file and compares values from this file with reference data.

With the DSE, we want to find the optimal trade-off between used hardware resources and the achievable performance. Therefore, FADSE can select values for the parameters in Table I. With 29 * 28 * 7 = 5684 possible configurations for the array and 3 * 9 * 8 = 216 configurations for the instruction cache, the design space comprises 1,227,744 ≈ 1.2 * 10^6 individuals in total for the GAP. Each configuration is evaluated with 10 integer-oriented benchmarks§ of the MiBench Benchmark Suite [20].

### 2.4. GAPtimize

One of the most important features of GAP is that the complete mapping of instructions, which could be understood as placement and routing, is carried out in hardware, hence without the need of using any special software to recompile or prepare a program for the execution on the GAP. This enables the execution of binary-compatible legacy code, for which a source code is not available.

For the evaluation of platform-specific code optimizations in this situation, a postlink optimizer has been developed. Our postlink optimizer is called GAPtimize. It works on statically linked executable files compiled with GCC for the portable instruction set architecture. GAPtimize is able to use information about the configuration of the target platform as well as performance data collected from a previous run of the program as feedback. Because of this, it can perform feedback-directed and adaptive code optimizations.

When multiple code optimizations are applied on a program, the basic questions are as follows: (i) which optimizations to apply, (ii) in which order they should be applied (phase-ordering problem, [21,22]); and (iii) which parameters for the optimizations should be chosen. These parameters have to be selected in respect of defined objectives, for example, the size of the executable or its performance on a given target architecture. In most production compilers, the selection and

---

§The 10 benchmarks are netw-dijkstra, auto-qsort, tele-adpcm-file-decode, offi-stringsearch, cons-jpeg-encode, cons-jpeg-decode, tele-gsm-encode, tele-gsm-decode, secu-rijndael-encode-nounroll, and secu-rijndael-decode-nounroll
Table II. Parameter space for the concurrent optimization of the parameters for hardware and code optimizations of GAP and GAPtimize.

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
<th>Size of domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>GAP</td>
<td>Array</td>
<td>29 * 28 * 6 = 4,872</td>
</tr>
<tr>
<td>GAP</td>
<td>ICache</td>
<td>1 * 1 * 1 = 1</td>
</tr>
<tr>
<td>GAP</td>
<td>Branch prediction</td>
<td>2 = 2</td>
</tr>
<tr>
<td>GAPtimize</td>
<td>Predicated execution</td>
<td>1 + 101 * 101 = 10,202</td>
</tr>
<tr>
<td>GAPtimize</td>
<td>Function inlining</td>
<td>1 + 1 * 101 * 101 = 10,202</td>
</tr>
<tr>
<td>GAPtimize</td>
<td>Static speculation</td>
<td>1 + 100 * 101 * 1 * 1 * 51 * 1 = 515,101</td>
</tr>
<tr>
<td>GAPtimize</td>
<td>Optimized replacement</td>
<td>2 = 2</td>
</tr>
<tr>
<td>GAPtimize</td>
<td>Phase ordering</td>
<td>2 = 2</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>≈ 2.1 * 10^{18}</td>
</tr>
</tbody>
</table>

order of optimizations are defined by optimizations levels such as -0.3 or -0.5 for the GCC. Parameters for single optimizations are mostly set to fixed values. Nevertheless, especially for GCC, it has been shown that it is possible to find both a better selection of optimizations and parameters with medium effort [23]. For the optimizations provided by GAPtimize, the mentioned issues have not yet been studied.

Currently, as optimizations for the GAP, a homebrew implementation of predicated execution, inlining of functions, a software-assisted replacement strategy for the configuration layers [24], and static speculation [25] have been implemented. Function inlining was in the focus of the exploration of parameters for hardware and code optimizations in our previous article [5]. The main goal was to find very good parameters in parallel for both the hardware parameters of the GAP and the heuristic used for inlining. This is extended now to all optimizations supported by GAPtimize.

The resulting parameter space—as described in Table II—comprises the seven hardware parameters of the GAP (array, instruction cache, if branch prediction shall be used; Section 2.3), a single parameter for every optimization automatically supported by GAPtimize to activate it or not (four in total), parameters for all the code optimizations (12 in total), and a single parameter to change the order in which function inlining and static speculation are used. When multiple optimizations can be applied, it is often a hard task to find a very good order to apply them as they influence one another, too. In our case, only inlining and static speculation can be interchanged, so the phase-ordering problem can be expressed as a single binary parameter. The hardware parameters have been restricted to 32 instead of 64 configuration layers and an instruction cache of limited size to boost the impact of the code optimizations. The parameter space comprises 4872 * 1 * 2 * 10,202 * 10202 * 515101 * 2 * 2 ≈ 2.1 * 10^{18} configurations in total.

When a configuration is evaluated, the average performance of nine benchmarks of the MiBench Benchmark Suite [20] is calculated. Because of the huge size of the parameter space and the significant time needed to evaluate a single configuration, a heuristic approach is the only useful way to approximate the Pareto front.

2.5. Objectives used

The quality of a configuration for the GAP and GAPtimize is mainly described by two objectives, that is, the used hardware resources and the performance that can be achieved.

2.5.1. Performance. The time that has to be spent on the execution of a program is a valid candidate to measure the quality of a processor’s configuration. For this, we configure our cycle-accurate simulator with the configuration generated by the DSE algorithm and run the simulation. It counts the total number of clock cycles for the simulation and the number of executed instructions.

Table III. Constants and their values to approximate hardware complexity of the GAP relative to a 32-bit integer ALU.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$h_{\text{ALU}}$</td>
<td>1.00</td>
<td>FU comprising an ALU</td>
</tr>
<tr>
<td>$h_{\text{LSU}}$</td>
<td>3.50</td>
<td>LSU</td>
</tr>
<tr>
<td>$h_{\text{l}}$</td>
<td>0.02</td>
<td>Configuration layer for a FU</td>
</tr>
<tr>
<td>$h_{\text{r}}$</td>
<td>0.02</td>
<td>Top register</td>
</tr>
</tbody>
</table>

ALU, arithmetic logic unit; FU, functional unit; LSU, load store unit.

These values are used to calculate the number of clocks per instruction (CPI), which is to be reduced. It is comparable because the number of instructions is the same with every run of the program and with any configuration of the processor.

For GAPtimize, which can change the number of instructions needed for executing a benchmark program, the number of clock cycles is divided by the number of instructions executed without using GAPtimize: clocks per reference instruction (CPRI).

### 2.5.2. Hardware complexity

The GAP is very scalable; hence, performance results have to be seen always in context with the used resources. Therefore, a model is needed to measure the hardware complexity of the GAP.

It is not possible to use bullet-proof numbers as there is not yet a hardware implementation for the GAP available. Hence, we introduce an approximation model. The aim of this model is as follows: (i) to be able to compare the hardware complexity of two configurations of the GAP and (ii) to compare the performance of two processors with different configurations but the same overall complexity. Hence, the purpose of the measure called complexity in the following is not to estimate the area used by the processor but to estimate the ratio between the complexities of two differently configured processors for comparison.

The complexity $H$ of the GAP is composed of the processor front end $H_{\text{front}}$, the array consisting of the ALUs, and the load store units with complexities $H_{\text{ALUs}}$ and $H_{\text{LSUs}}$, the instruction cache with complexity $H_{\text{ICache}}$, and some other components that exist exactly once in the GAP and are independent of parameters. These components contribute $H_{\text{constant}}$; the data cache\(^1\) is one of them because it is not configurable at the moment. $H$ can be defined as

$$H = H_{\text{front}} + H_{\text{ALUs}} + H_{\text{LSUs}} + H_{\text{ICache}} + H_{\text{constant}}$$

As we need the complexities only for comparability, we ignore $H_{\text{front}}$ and $H_{\text{constant}}$ in the following because they are the same for all configurations of GAP.

To be able to calculate the complexities of the FUs in the array, we declare the constants in Table III. Appropriate values for these constants are found with the following three approaches:

- The thermal simulation tool *HotSpot* [27] comprises a floor plan for the 64-bit Alpha 21264 processor [28]. This floor plan was gained from carefully taking the measures from a die photo from this processor.
- *Gupta et al.* presented in their article [29] numbers for several parts of a processor and tried to find process-independent numbers to estimate their size.\(^{**}\)
- The combined simulator for power, area, timing, and temperature (*McPat* [30]) is shipped with a configuration for the 64-bit Alpha 21364 processor, which extends the Alpha 21264. The area approximation is based on the work by Gupta *et al.* [29] and Rodrigues [31].

From these approaches, we isolated the size of a 64-bit ALU, a 64-bit register, and 64 kB of instruction cache. As the GAP is a 32-bit processor, we divided the size of the ALU and the register by two; this method is supported by for example Gupta *et al.* [29]. The numbers are finally normalized to the size of an integer ALU (so, $h_{\text{ALU}} := 1$ for all sources); Table IV shows the results.

\(^{1}\)Details on the memory hierarchy are described in [26].

\(^{**}\)The standard deviation of the values they used is quite high.
The results for a 32-bit register vary quite a lot between the different data sources. Nevertheless, the cost of a register is for all sources very small compared with the cost of a 32-bit integer ALU. Because of this, we set $h_{\text{ALU}} := 0.02$. We choose as additional cost for a configuration layer the same cost $h_{\text{ALU}} := 0.02$ although the required memory is double the size but of a much lower complexity. For a load store unit, numbers were available only from McPat and the floor plan vary between around 1.5 and 5.4 times the size of an integer ALU. We define $h_{\text{LSU}} := 3.5$.

The cost of a 64-kB instruction cache compared with an integer ALU is around 16.5 on average for the three data sources. CACTI [32] calculates a total area of 5.52 mm$^2$ for a cache with parameters similar to those of the Alpha processors.†† To be able to use CACTI to dynamically approximate the complexity of the instruction cache $H_{\text{ICache}}$, we set 16.5 equal to an instruction cache with a total area of 5.52 mm$^2$. We calculate the approximated area of a cache for the GAP with CACTI and then multiply it with 16.5 to get its complexity compared with an ALU.

With these constants, we can define the complexities $H_{\text{ALUs}}$ and $H_{\text{LSUs}}$. In both formulas, where $C_c$ is the number of columns, $C_r$ is the number of rows, and $C_l$ is the number of configuration layers of the array of FUs, we first calculate the complexity of the units and then add the complexity caused by the configuration layers:

$$H_{\text{ALUs}} = (C_c * h_{\text{ALU}} + C_r * C_c * h_{\text{ALU}}) + C_r * C_c * C_l * h_{\text{ALU}}$$

$$H_{\text{LSUs}} = C_r * h_{\text{LSU}} + C_r * C_l * h_{\text{LSU}}$$

For a GAP with an array of 12 lines and columns, 32 configuration layers, and 8-kB instruction cache, the hardware complexity $H$ is computed as follows:

$$H = H_{\text{ALUs}} + H_{\text{LSUs}} + H_{\text{ICache}}$$

$$H_{\text{ALUs}} = (12 * 0.02 + 12 * 12 * 1.00) + 12 * 12 * 32 * 0.02 = 144.24 + 92.16 = 236.40$$

$$H_{\text{LSUs}} = 12 * 3.50 + 12 * 32 * 0.02 = 49.68$$

$$H_{\text{ICache}} = 0.856 \text{ mm}^2 * 3 \text{ mm}^{-2} = 2.57$$

$$H = 236.4 + 49.68 + 2.57 = 288.65$$

In this work, our developed FADSE has two distinct objectives: to minimize the clocks per instruction (or CPRI, depending on the context) and to minimize the global hardware complexity ($H$).

### 3. EVALUATION

The evaluation is split into two parts. First, in Section 3.1, an ADSE of the hardware parameters of the GAP is performed with the algorithm NSGA-II. An in-depth analysis of the found high-quality configurations is performed, showing relations between the parameters and how to choose them according to the available hardware complexity. Also, it is explained how hardware complexity is calculated.
should be distributed on the different components of the GAP to achieve good results. Beyond this, the influence of the population size for NSGA-II is evaluated.

Second, an ADSE of the parameters of GAP together with those of the postlink optimizer GAPtimize (Section 2.4) is performed by evaluating the influence of code optimizations on efficiency and performance.

3.1. Automatic design space exploration on Grid ALU Processor

We choose NSGA-II [7], being one of the most used and cited heuristic algorithm [33] in computer science, as algorithm for the DSE and run it with FADSE on the design space as described in Section 2.3. The population size is set to 100, the crossover probability to 0.9, and mutation probability to 1/(number of parameters) as recommended [7]. All other parameters were set as recommended by Deb et al. [7], too. We have used single-point crossover and bit flip mutation operators. The selection operator is binary tournament as proposed by Deb et al. [7].

3.1.1. Performance versus hardware complexity. As the hypervolume in Figure 3 shows only very small improvements after about 50 generations, the DSE was stopped, and we obtained the results in Figure 4(a,b). The whole design space with all evaluated configurations is shown in Figure 4(a). The maximum hardware complexity of a configuration is about 10,000. The configurations that are most to the left and to the bottom have the highest quality (both objectives need to be minimized) and form the Pareto front approximation.

The artifact at the right side of Figure 4(a) shows configurations with an extremely big instruction cache. They can reach the optimal performance but are, because of the huge cache size, not area effective at all.

There is basically no performance improvement for configurations with hardware complexity above about 1200, so Figure 4(b) shows the more interesting section of the total result space where performance improves when hardware complexity is increased, too.

So that the highest possible performance is reached, a hardware complexity of about 1000 is necessary. If complexity is further increased, then performance is improved only very marginally. A configuration of the GAP with a complexity of about 1000 can have 32 lines, 11 columns, 32 configuration layers, and 512-kB instruction cache (32 × 11 × 32, 512 kB).

If a performance cutback of 10% is tolerable, then the hardware complexity can be reduced by 66% to about 330. The necessary hardware complexity to reach a performance 20% below the optimum is 170, that is, hardware complexity can be reduced by 83% for a performance reduction of 20%. Hence, with very little hardware effort, a still quite high performance can be achieved. Nevertheless, performance breaks down if hardware complexity is further decreased.

The configurations found by FADSE are superior to the manually found reference configurations defined by Shehan et al. [19]. With the automatically found configurations, typically the same performance can be achieved with a much smaller hardware effort (for details, see [5]).

The so far presented approximation of the Pareto front for the GAP shows basically which maximal performance can be reached with the GAP and how performance is related to hardware effort. But it does not give information on how to configure the GAP so that the deployed hardware
resources can be used effectively. With the notation described in Section 2.1, only the position of very good configurations in the objective space is known, but not their representation in the parameter space.

Hence, all configurations evaluated by FADSE are carefully examined. First, they are separated into two classes, that is, perfect and good, as already shown in Figure 4(a,b). All configurations...
AUTOMATIC OPTIMIZATION OF PARAMETERS FOR HARDWARE AND SOFTWARE

(a) Complexity of the functional units

(b) Complexity of the configuration layers

(c) Complexity of the instruction cache

(d) Size of the instruction cache (kB)

Figure 5. Distribution of the total hardware complexity of configurations (x-axis) to the different components of the GAP (y-axis). Very good configurations are depicted in yellow/gray; all other configurations are marked by a ×.

with distance‡‡ of at most ε to their closest configuration of the interpolated approximation of the Pareto front (Figure 4(c)) are classified as perfect. All other configurations are called good. We selected ε so that about 24.5% of the all 3209 configurations are ranked perfect. The perfect configurations are also the most efficient configurations because they gain very high performance from the hardware complexity.

3.1.2. Optimal distribution of complexity. First, we want to show how a given budget of hardware complexity should be distributed on the different hardware elements. In Figure 5(a–c), this is shown for the FUs, the configuration layers, and the instruction cache. Theoretically, the complete complexity could be spent on FUs only or the instruction cache could be overweight. For a visualization of this in all three images, there is a dashed line; if all complexity is used for the displayed component, a symbol would be on this line. ◦ is used to represent perfect configurations, and all other configurations are depicted by ×. From the images (Figure 5(a–c)), it can be derived that for a high fraction of the found perfect configurations, about 50% of the total complexity should be spent for FUs, about 40% on the configuration layers, and the rest, a budget of around 100, on the instruction cache. The size of the instruction cache (Figure 5(d)) is, for nearly all configurations, at most 1024 kB.

3.1.3. Optimal selection of parameters. An in-depth analysis of the parameters of the array of the GAP is shown in Figure 6(a–d). Perfect configurations are again shown by ◦, and all other configurations are depicted by ×. In Figure 6(c), it is easy to see that the number of configuration layers is, for perfect configurations, rapidly increased until a total hardware complexity of 200; after this,

‡‡Details on the distance measure are explained in [14].
Figure 6. Parameters of the array for different hardware complexities (x-axis); very good configurations are depicted in yellow/gray; all other configurations are marked by a ×.

most perfect configurations have $2^6 = 64$ configuration layers with some outliers to $2^5 = 32$. The number of columns for perfect configurations is increased almost linearly with a small curve to the top starting at eight for a hardware complexity of 50 to the upper limit 32 for a hardware complexity of 1200. Nevertheless, the values are scattered more and more with increasing complexity. The columns are selected with a little scatter starting at four for small complexities to about 14 for hardware complexity of 1200. The upper limit of 32 columns is never reached until complexity of 1200, and—as mentioned already—performance does not improve with complexities over 1200. As conclusion, more than 16 columns (the observed maximum for any perfect individual) cannot be used effectively by the GAP. The ratio of lines per column for perfect configurations starts at about one line per column for very small configurations and quickly moves to about two lines per column starting at hardware complexity of 400.

Table V. Some very efficient/effective configurations for the GAP.

<table>
<thead>
<tr>
<th>Lines</th>
<th>Columns</th>
<th>Layers</th>
<th>ICache (kB)</th>
<th>Complexity</th>
<th>CPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>4</td>
<td>2</td>
<td>8</td>
<td>62.78</td>
<td>1.2863</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>2</td>
<td>16</td>
<td>63.54</td>
<td>1.0395</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>2</td>
<td>64</td>
<td>66.04</td>
<td>0.8746</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>32</td>
<td>64</td>
<td>90.04</td>
<td>0.6872</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>64</td>
<td>256</td>
<td>186.11</td>
<td>0.6051</td>
</tr>
<tr>
<td>18</td>
<td>7</td>
<td>64</td>
<td>128</td>
<td>383.22</td>
<td>0.5555</td>
</tr>
<tr>
<td>23</td>
<td>9</td>
<td>64</td>
<td>128</td>
<td>591.84</td>
<td>0.5394</td>
</tr>
<tr>
<td>32</td>
<td>11</td>
<td>64</td>
<td>512</td>
<td>1005.47</td>
<td>0.5243</td>
</tr>
</tbody>
</table>
The configurations in Table V are supposed to be very effective in using hardware complexity, that is, the best performance can be gained from the hardware complexity.

3.1.4. Variation of the parameters of (NSGA-II). For every generation of size \( n \), the NSGA-II algorithm normally knows \( n \) already evaluated individuals, the parent generation. They are used to generate with crossover and mutation \( n \) new individuals called offspring, which have to be evaluated. When this is completed, the best \( n \) individuals (taking into consideration the domination relationship and a density function) are selected from the union of the parent generation and the offspring.

To evaluate the influence of the population size, which is an important factor for the total duration of a DSE process, we set it to different values (12, 24, 50, and 100). For 12 and 24, we were not able to reliably find good results because the selection/mutation did not work effectively. To compare the progress made with generations of size 50 and 100 in respect to the number of evaluated individuals, we calculated the hypervolume and coverage for each generation. The difference in hypervolume is less than 1% (only marginal). Coverage (Figure 7) does not show a clear winner, too. A large population clearly has the benefits of a better exploration, but it will also mean more simulations.

The selection process of the individuals can be observed in Figure 8. The number of new individuals, that is, the individuals that have never been generated before in the exploration, decreases over time, for example, from 100 to 35. Because of this fact, it is very effective to save results for individuals for reuse for example in a database. This can speed up the DSE very much. In a run with 100 generations, a total reuse factor of 67% was observed.

The number of individuals that are better than their parents and hence survive for the next generation decreases over time, too. According to Figure 8, at the beginning of the exploration, many of the offspring individuals are better than their parents; therefore, they move into the next generation. As the algorithm progresses, fewer good individuals are found. This is adequately correlated with the hypervolume, whose improvement from one generation to the next decreases (Figure 3).

3.2. Automatic design space exploration on Grid ALU Processor with GAPtimize

To explore the design space of GAP and GAPtimize with FADSE, we chose the SMPSO algorithm [9]. In our previous studies, we saw that SMPSO tends to have a better convergence speed [3], although the final results obtained by NSGA-II and SMPSO are very similar. Because GAPtimize...
brings an extra effort to the simulation compared with the previous experiment (that used only GAP), hence, a reduction of the time required for the DSE was needed. A distributed version of the algorithm has been implemented, which allows a parallel evaluation of configurations. Optimizations were also implemented to avoid repeating simulations for the same inputs.

A swarm size of 100 individuals was used during the ADSE process. The exploration was stopped after 20 generations as the hypervolume value did not show significant improvements any more. During these generations, a total of 1682 individuals were simulated, representing a very small fraction of the total design space (far less than 1%).

To evaluate the improvements brought by GAPtimize, we performed a comparison against the results obtained during another ADSE process performed only on the hardware parameters without GAPtimize. Figure 9 shows the approximations of the two Pareto fronts.

At first glance, the two curves are very close together. Especially starting at a hardware complexity of about 600, almost no difference is noticeable. The reason is that the impact of the code optimizations is smaller when more hardware resources are available, for example, a larger array and a higher number of configuration layers decrease the influence of function inlining and other optimizations. Also, not all the benchmarks are sensitive to the same set of optimizations. We show average results, so if only a small number of benchmarks is influenced by an optimization, they will not necessarily have a major impact on the numbers presented in the end.

For lower hardware complexities, the influence of code optimizations is much higher. Both of the following goals were met: (i) increasing the performance while maintaining the same hardware complexity (cost) and (ii) finding configurations with lower hardware complexity while preserving the performance. The results are shown in Figure 10(a,b). Although in general, very good configurations are found by the heuristic algorithm, because of the immense size of the design space and the partially negative effect of the code optimizations, it is not successful in some situations. This is the case especially for hardware complexities between 650 and 750. A superior solution could have been found if all optimizations were deactivated, resulting in no acceleration. But this is in contradiction to nearly all other near-optimal configurations and hence hard to find. Nevertheless, the negative acceleration is only very little.

The maximum acceleration of 15% is achieved for a configuration with a hardware complexity of about 200, which is a very small configuration of the GAP. In this region, the effects of the code optimizations are clearly visible. For a hardware complexity of up to 550, the acceleration is often higher than 6%.

When the code optimizations are active, the hardware complexity can be reduced for configurations with a CPRI of up to 1.2, often by more than 21%, without losing performance. In the best case, a processor with a reduced hardware complexity of 52% shows the same CPRI. Thus, the code optimizations can reduce the hardware requirements a lot while keeping the same performance.
(a) Performance improvement at the same hardware complexity

(b) Reduction of the necessary hardware complexity at the same performance

Figure 10. Effects of the code optimizations implemented in GAPtimize on the execution of programs on the GAP.

A detailed analysis of the configurations of high quality revealed that for most very good configurations the branch prediction of GAP is used in combination with predicated execution. With increasing complexity, the use of predication is reduced a bit. Inlining and static speculation are applied for most of the configurations; usually, one or more of these optimizations are active. If both are selected, then inlining is performed prior to static speculation. This behavior is not changed with growing complexities. The optimized replacement strategy quick drop LRU (qdLRU) is valuable for configurations of all complexities, too.

4. RELATED WORK

4.1. Automatic design space exploration with increased level of parallelism

With respect to FADSE, there are existing tools that try to address the problem of ADSE in the domain of processor architectures. One of them is Magellan [34], which focuses on a multicore architecture. From our point of view, the main drawbacks of this tool are that it is bound to a certain simulator (SMTSIM) and that it is not multi-objective in a true way. The user can set a boundary for its power/area ratio but cannot explore the entire Pareto front in a single run.

Archexplorer.org [35] is a collaborative web site where users can upload components of processors contributing to a DSE with the goal of finding an optimal processor. The used algorithms cannot be controlled, and the tool is strongly related to UNISIM (at the moment); hence, it cannot be used for a different processor simulator easily.

Non-ad hoc search algorithm [36] is similar but provides only self-developed single-objective algorithms. M3Explorer [37] offers a variety of multi-objective DSE algorithms, but it lacks the distributed simulation. Nevertheless, M3Explorer is a sophisticated tool and can also be configured for different domains with reasonable effort.

As conclusion, even though parallel algorithms exist and although sequential evolutionary algorithms can be parallelized easily, none of the presented tools shows this ability. This and its configurability are strong advantages of the improved version of FADSE introduced in this article to reduce the time needed for a DSE dramatically. FADSE also proved to be a robust and adaptable tool as it has been used for various DSE experiments on single-core, simultaneous multithreading, multicore, and system-on-a-chip architectures [4, 38].
4.2. Design space exploration of the GAP

The presented DSE based on [5] is also the first multi-objective one for this processor. So far, only Shehan et al. [19] have tried to find good points for the parameters of the array of the GAP, but the instruction cache was fixed, a manual approach was used, and only a very little part of the design space was evaluated, resulting in a high risk to find local minima instead of global optima.

Similar to the exploration of the hardware parameters of the GAP is (for example) the work by Sheldon and Vahid [39], who selected the parameters for a soft core on a field programmable gate array as a way to achieve the best configuration for a given program.

4.3. Design space exploration of hardware and software parameters

For the given setup consisting of GAP and GAPTimize with multiple code optimizations, a DSE has not been run so far. The most similar articles are about the following: (i) the parallel optimization of hardware and software parameters; (ii) automatic selection of compiler optimization; and (iii) automatic configuration of compiler optimizations.

Agosta et al. [40] did a multi-objective search for hardware parameters in combination with code transformations for a configurable soft core on a field programmable gate array. The parameters are chosen to be optimal for a group of benchmarks. As objectives, energy consumption and runtime are used. It is also explained why it is not sufficient to first optimize the hardware parameters and as a second step to look for optimal code optimizations and that this has to be performed in parallel to really reach the best results.

In the article by Monsifrot et al. [41], machine learning is used to optimize compiler heuristics. The main example is loop unrolling, that is, a model shall predict, from loop characteristics, if the impact of unrolling is probably going to be positive. Stephenson et al. [42] also applied machine learning techniques to improve compiler heuristics, too.

With an algorithm derived from SPEA2, Hoste et al. [23] gathered new optimization levels comparable with the well-known switches -O3 or -Os for GCC. Their optimization sets show better results also for unknown programs, so they can be generally used. Almagor et al. [43] have similar targets but optimized the execution of only a single program comparable with adaptive optimization.

The DSE of the parameters of GAPTimize is basically an iterative approach, so optimizations are applied step by step. In this context, the work of Knijnenburg [44] should be mentioned, too. Taking worst-case execution time into account, Lokuciejewski and Marwedel [45] have explored compiler optimizations for real-time systems.

5. CONCLUSION

In the development process of novel processor architectures, it is important to find optimal parameters for both the new processor and code optimizations because their influence can be different if the underlying architecture is changed. Often, more than a single objective has to be optimized.

This article mainly addressed two topics. First, the multi-objective FADSE is extended. FADSE has been developed to intelligently explore relevant subspaces of a huge design space using state-of-the-art evolutionary and bio-inspired search algorithms. Its focus is mainly on processor architectures but can be changed or extended to other domains easily. The duration of DSEs can be significantly decreased by evaluating configurations in parallel on multiple cores, machines, or a high-performance computing system. To cope with errors in the network or in the evaluation of a configuration, we introduced mechanisms to strengthen the robustness of FADSE. This is performed by monitoring simulations continuously, comparing the results with reference data, and restarting them automatically on another computing node for a second try if a node is assumed to be erroneous.

Second, we showed that the improved version of FADSE is able to thoroughly explore the vast design space of the GAP, a novel research processor to accelerate sequential instruction streams, and its postlink optimizer GAPTimize for target-specific whole-program code optimizations. System performance and an approximation of the hardware complexity of the elements of the GAP with variable size were used as objectives. Hardware complexity is calculated in comparison with an integer ALU.
The result of the DSE is first of all a description of the relation between performance and hardware complexity, so the interaction between the objectives is described. In the ADSE, only 0.29% of all possible configurations are evaluated by NSGA-II with 10 benchmarks per configuration. The approximation of the Pareto front found by the ADSE shows that there is no performance increase with hardware complexities above 1000. A performance of 10% below the maximum value can be reached with a hardware complexity of 330. Below a hardware complexity of 100, performance starts collapsing.

Third, relationships between the hardware complexity and the hardware parameters of very good configurations were revealed. For very good configurations, the number of rows of the GAP array should be increased linearly from five for low complexities to 32 rows for configurations with a hardware complexity of about 1000. Also, the array should have about two rows per column, hence never reaching the maximum value of 32. The number of configuration layers is set to the maximum value $2^6 = 64$, starting at complexity 200. Very good configurations tribute about 50% of their total complexity to the FUs, as much as possible to the configuration layers. The remainder is used for an instruction cache of typically 256 to 1024 kB. This shows that complexity can be used more effectively if spent on configuration layers instead of instruction cache.

Fourth, FADSE turned out to be able to cope with hardware and software parameters in parallel, too, although the design space is huge with about $2.1 \times 10^{18}$ individuals. In this setup, GAP and GAPtimize were used, and for optimal parameters, the following were searched: (i) the optimal subset of four optimizations; (ii) the order of two optimizations; and (c) heuristics of three optimizations. The configurations found by FADSE show a maximum reduction of the total execution time of 15% compared with using GAP alone. The maximum reduction of the hardware complexity sustaining the same performance by using code optimizations is 52%.

As future work, we propose to further extend FADSE with possibilities to reuse existing results whenever possible. Together with the ability to boost the DSE process as described in article [14], this could be a perfect tool for adaptive compilation for generic platforms, that is, finding the best settings for compiler options and the best parameters in a way that a given program can be executed with high quality (e.g., performance) on a given platform. It would be interesting to do such attempts not for the GAP but for common architectures such as x86 and embedded architectures, too.

ACKNOWLEDGEMENTS

Ralf Jahr is working for the GAP Project funded by the German Research Foundation (DFG). Lucian Vintan was partially supported by CNCSIS no. 485/2008 research grant offered by the Romanian National Council for Academic Research. Horia Calborean was supported by POSDRU financing contract POSDRU 7706.

REFERENCES


