

# CONFIGURATION CODE GENERATION AND OPTIMIZATIONS FOR HETEROGENEOUS RECONFIGURABLE DSPS

Suet-Fei Li, Marlene Wan and Jan Rabaey

Berkeley Wireless Research Center and the Department of EECS  
University of California at Berkeley

**Abstract - In this paper we describe a code generation and optimization process for reconfigurable architectures targeting digital signal processing and wireless communication applications. The ability to generate efficient and compact code is essential for the success of reconfigurable architectures. Otherwise, the overhead of reconfiguring could easily become the system bottleneck. Our code generation process includes the evaluation a set of tradeoffs in system design, software engineering as well as usage of a set of local and global optimization techniques. By doing so we are able to achieve results of significantly lower overhead.**

## I. INTRODUCTION

Future networked embedded devices will need to support multiple standards of communication and digital signal processing. Reconfigurable systems have recently emerged as a promising implementation platform for such embedded computing by exhibiting both high performance [1] and low power [2][3] frequently required by such system-on-a-chip designs. The current trend of reconfigurable architectures in both general purpose computing and embedded digital signal processing is to combine a programmable processor with reconfigurable computing components of different granularities (fine-grain [4][5], datapath [6] and mixed [7]). The problem of interface generation between hardware and software has recently gained significant attention by the VLSI CAD community. The problem of integrating processors with reconfigurable elements has added another dimension to the interface generation problem - between software and configware. Careful configuration and interface code generation is essential [8] to ensure that the overhead of reconfiguring will not offset the speed and energy savings of reconfigurable components. This need is especially pronounced when the reconfiguration frequency is large within an application and when the timing constraints on the application are tight – which is often the case for real-time DSP and communication applications.

In this paper, we present a code generation and optimization process for reconfigurable architectures targeting digital signal processing applications. While the concept of our code generation and optimization process is machine-independent and can be applied to any reconfigurable architecture, we show the effectiveness of our code generation process on a heterogeneous reconfigurable

digital signal processor (Pleiades architecture template [9]). Similarly, while the process can be utilized as the backend to any system compilation tool, we use the software methodology proposed in [10] as the front end to obtain a good software-configurable partitioning. In the remainder of this paper, we will first introduce an overview of the architecture template and the software methodology in Section II. A detailed description of the proposed code generation and optimizations is given in Section III. At the end of this paper, we illustrate the effectiveness of this process by providing results for a baseband voice processing architecture.

## II. BACKGROUND

A heterogeneous reconfigurable digital signal processor template is shown in Figure 1. The template consists of a microprocessor and computational coprocessors of different programming granularities (referred to as satellites in the rest of this paper). The computation model on the processor is shown in Figure 2. A single thread can spawn computations on a cluster of satellite processors. The underlying satellite processors can support reconfigurations so multiple spawning within an application is possible (right side of Figure 2). From the programmer's point of view, reconfiguration means the configuration registers corresponding to each satellite (to specify the operation of the satellite) and the reconfigurable interconnect (to connect a cluster of satellites). Each one of the configuration registers is part of the microprocessor's memory map.

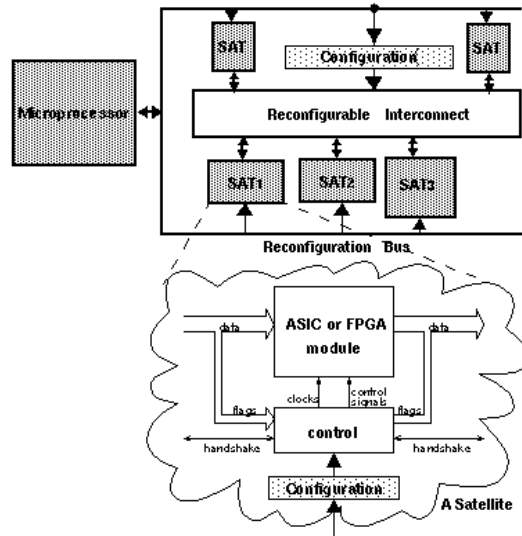


Figure 1 Heterogeneous Reconfigurable Architecture Template (Pleiades Architecture)

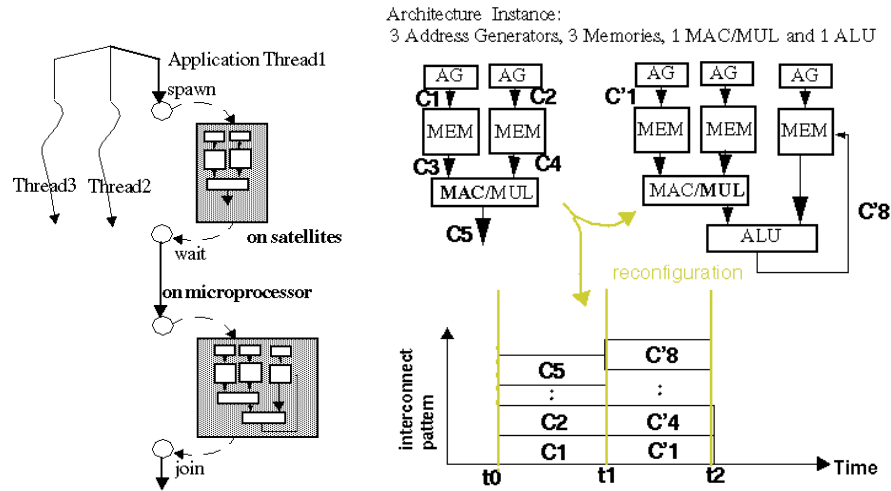


Figure 2 Model of Computation and Reconfiguration

Since the compilation of a single thread is the basis of other system-level scheduling tools that can utilize multiple threads, we concentrate our efforts in generating efficient interface code for a single application thread. For the above architecture template, the interface overhead mainly consists of the reconfiguration of satellites and the synchronization between the microprocessor and the satellites. In General, reduction of the overhead can be achieved by generating clever software or replacing parts of the software by special hardware dedicated to reconfiguration. In this paper, we focus on the software solution and discuss hardware enhancements in the future work section.

### III. CODE GENERATION AND OPTIMIZATIONS

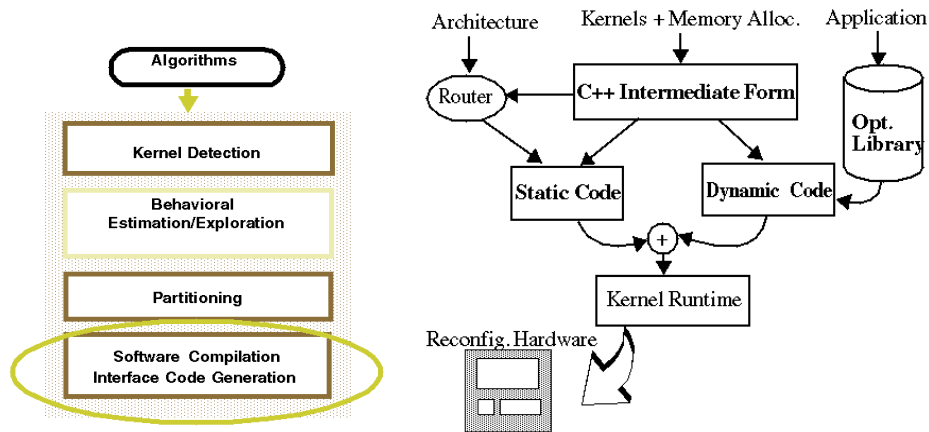


Figure 3 Software Methodology and the Backend Code Generation

## 1. Overall Software Methodology and the Intermediate Form

The overall software flow is shown on the left of Figure 3. The input to the software flow is an algorithm specified in C/C++. After mapping and partitioning across different architectures, the program body to be executed on the microprocessor remains as C code. Computation performed on satellites (this kind of computation will be referred to as kernel in the rest of the paper) is encapsulated in a procedure call and described in a high-level (currently in C++) intermediate form.

Our goal is to generate efficient and compact configuration and interface code from the C++ intermediate form based on the rest of the program structure and the underlying architecture. It is achieved by following the steps presented on the right of Figure 3. Based on the kernel structures specified by the intermediate form, the baseline code is generated. The architecture and the application program are then used to trigger a set of optimizations to improve upon the baseline code. We will first introduce the concept of the intermediate form in the remainder of this section. The basic code generation process is presented in Section III.2 and optimizations are discussed in Section III.3 and III.4.

The intermediate form is based on the concept of processes (satellites) and queues (connection between satellites). Since the intermediate form has all satellite functionality and connectivity information (shown in Figure 4), it is equipped with configuration and interface code generation capability. A code generation library (corresponding to each satellite as well as the whole kernel) is provided to serve the purpose of automatic code generation. The library currently contains all the basic building blocks (satellites) in Pleiades: MAC/MUL, ALU, SRAM, AGP (address generator), INPUT\_PORT and OUTPUT\_PORT etc. The kernel specified in the intermediate form is like a procedure call – the structure is fixed for each kernel but each invocation of the kernel can pass in different parameters.

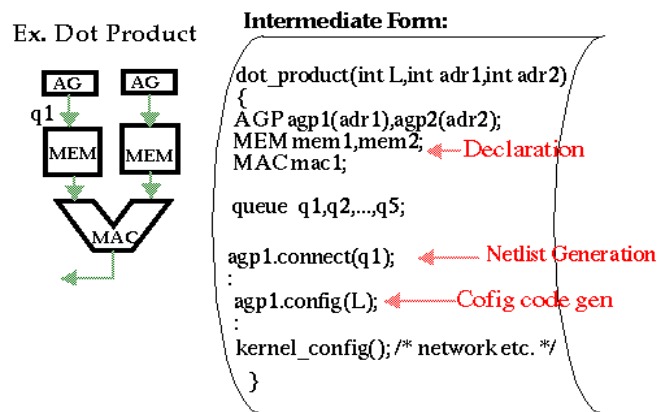


Figure 4 Kernel Mapping and its Specification in the Intermediate Form

## 2. Basic Code Generation Techniques

If the configuration and interface code can be determined at compile time, it is defined as static, while if it can only be determined at runtime, it is dynamic. It is more efficient to execute static code. Based on functionality, the different types of configuration code are divided into three categories. Each category has its unique characteristics that help us determine if it favors static or dynamic configuration:

- **Interface code:** This code takes care of synchronization between the processor and satellites, including resetting of all satellite processors and network before kernel execution. Due to memory and performance constraints, an off-the-shelf operating system for the microprocessor is not used. Instead, a more streamlined version of the operating system is generated. These codes are determined statically.
- **Configuration code for the satellites:** For a given kernel, most of the operations performed on satellites are known, and therefore are statically generated. For example, the configurations of SRAM, ALU and MAC/MUL are fixed for each kernel. Only the configuration of AGP varies from one invocation of kernel to another, and therefore has to be passed in as parameters from the main application program. Hence, configuration in the former case is static and in the latter dynamic.
- **Configuration code for the reconfigurable interconnects:** It is very time consuming to perform runtime routing. Therefore, the netlist is routed on the underlying architecture using the router described in [10] at compile time.

Therefore, the configuration is static.

While for a single kernel mapping, the baseline code sequence is determined according to the above basic criteria, generating efficient, yet modular, interface and configuration code for kernels embedded within a larger application involves many more tradeoffs and optimizations which we will describe in the following sections.

## 3. Trade-off and Local Optimizations

A couple of tradeoffs are carefully evaluated to produce a good executable code sequence. First, while minimizing total power consumption is the principal goal, DSP and wireless systems have limited memory size, which constrains the total code size of the application program. We analyze the tradeoff between the two factors – application performance and code size, to determine a right modularity for the generated code. The second trade-off is the reusability of the code versus performance/power. One must customize configuration code for different kernels and applications to achieve satisfactory performance while balancing the development effort and code complexity. Note that in this paper, we translate the optimization of reconfiguration power consumption and performance into the minimization of total number of reconfiguration cycles on the microprocessor.

**Performance/Power vs. Code Size trade-off.** Intuitively speaking, a program that takes the fewest number of cycles to execute would be a “flat” program,

meaning that no modularity is present and therefore kernel procedure calls are in-lined and expanded (in loops). However, a simple application has thousands of kernel procedure calls and each kernel has up to 100 lines of straight code. Clearly, the sheer code size makes this approach infeasible and hence some modularity in the executable code is required. In most digital signal processing and communication applications, there are only limited number of kernel types (dot\_product and FIR in speech coding, DCT in image coding etc.) with different parameters (vector length, starting and ending address in memory). Therefore, we made each kernel type its own procedure call in the generated code. Within a kernel, dynamic and static interface and configuration code of the individual satellite is simply straight code expansion to avoid procedure call overhead.

By following the above procedure, we were able to keep the memory requirements of the program manageable without significantly sacrificing much of the power/performance. For example there are totally about 100 thousand kernels in VSELP. The code size is decreased by four orders of magnitude while the performance is compromised by 50% compared to a flat code sequence (see Table 1). There are certainly other points in the performance versus code size curve but we will mainly concentrate on code optimizations specific to reconfigurable systems in this paper.

Code generation schemes	Normalized Configuration Time (total # of cycles)	Normalized code size
Flat	1	1
Modular	1.56	$7.63 \cdot 10^{-5}$

Table 1. Comparison of memory and performance between the flat and modular code sequence for the VSELP Application

**Generality vs. Performance/Power tradeoff.** The second tradeoff is code generality vs. performance/power. It is desirable to have customized code routines to take advantage of the special properties of the target application and the kernels involved. However, we also want to maximize code reusability. Significant effort altering the existing library should not be required to achieve acceptable performance.

System bottleneck analysis indicated that address generator configuration is the dominating factor during kernel configuration. This could be explained by the fact that its configuration has the most dynamic component. For a given kernel, only parts of the AGP configuration register fields are relevant and the rest could be treated as don't care. Thus it is advantageous to provide customized AGP configuration routines for the different kernels.

The generality vs. performance/power trade-off issue is also present when we are dealing with the communication from the satellites to the microprocessor. In order to handle applications with multiple parallel threads, we provide the interrupt driven communication primitive in our interface library. That is, the coprocessors

interrupt the embedded microprocessor when they want to communicate. This way communication in a multi-threading environment could be orchestrated in a clean and organized way. Note that sophisticated communication primitives such as interrupt handling and polling do have significant overhead. While for applications that have lots of parallelism to exploit this overhead might be necessary. For applications that are sequential in nature, this expensive overhead cost might not be worthwhile to pay. In such case, simpler and cheaper primitive might be more suitable. For example, the VSELP algorithm is mostly sequential, so the sleep and wakeup power saving mode of the processor is sufficient to implement the communication primitive. In this simple primitive, the microprocessor goes to sleep after it finishes configuration and triggers Pleiades to start running, then Pleiades wakes the microprocessor up when it is done. By replacing the interrupt scheme by the sleep-and-wake communication primitive, significant savings are achieved.

#### 4. Global Optimizations

Besides performing optimizations at the kernel level by choosing the right modularity and providing optimized code libraries, we also exploit the special features of the underlying reconfigurable architecture and the application structure to generate better code sequence.

**Program Cache.** Multiple-context FPGA [12] has been proposed as one of the architecture solutions to reduce configuration cycles in between reconfigurations. Our underlying architecture also supports limited multiple-context configurations. In particular, the AGP instruction registers are deeper and can support up to 5 contexts. Since kernels within DSP applications have only limited instruction patterns, all AGP instructions are often stored in the reconfigurable satellites without reconfiguration from the processor.

**Partial Reconfiguration.** When two identical kernels are called sequentially, only part of the configuration data has to be loaded into the satellites. Specifically, since a particular kernel has a fixed satellite cluster structure and fixed operations, the only configurations that have to be reloaded are the interface code and AGP configurations (for possibly different starting and ending address, vector length etc.). We use the SUIF [13] compiler front-end to discover if a kernel procedure is in a nested loop or identical kernels appear consecutively in the same control flow.

#### 5. Example

To better illustrate the global and local optimization techniques mentioned above, the dot product kernel will be used as a simple example (see Figure 4). The following figure shows the pseudo code for the configuration routine of the kernel.

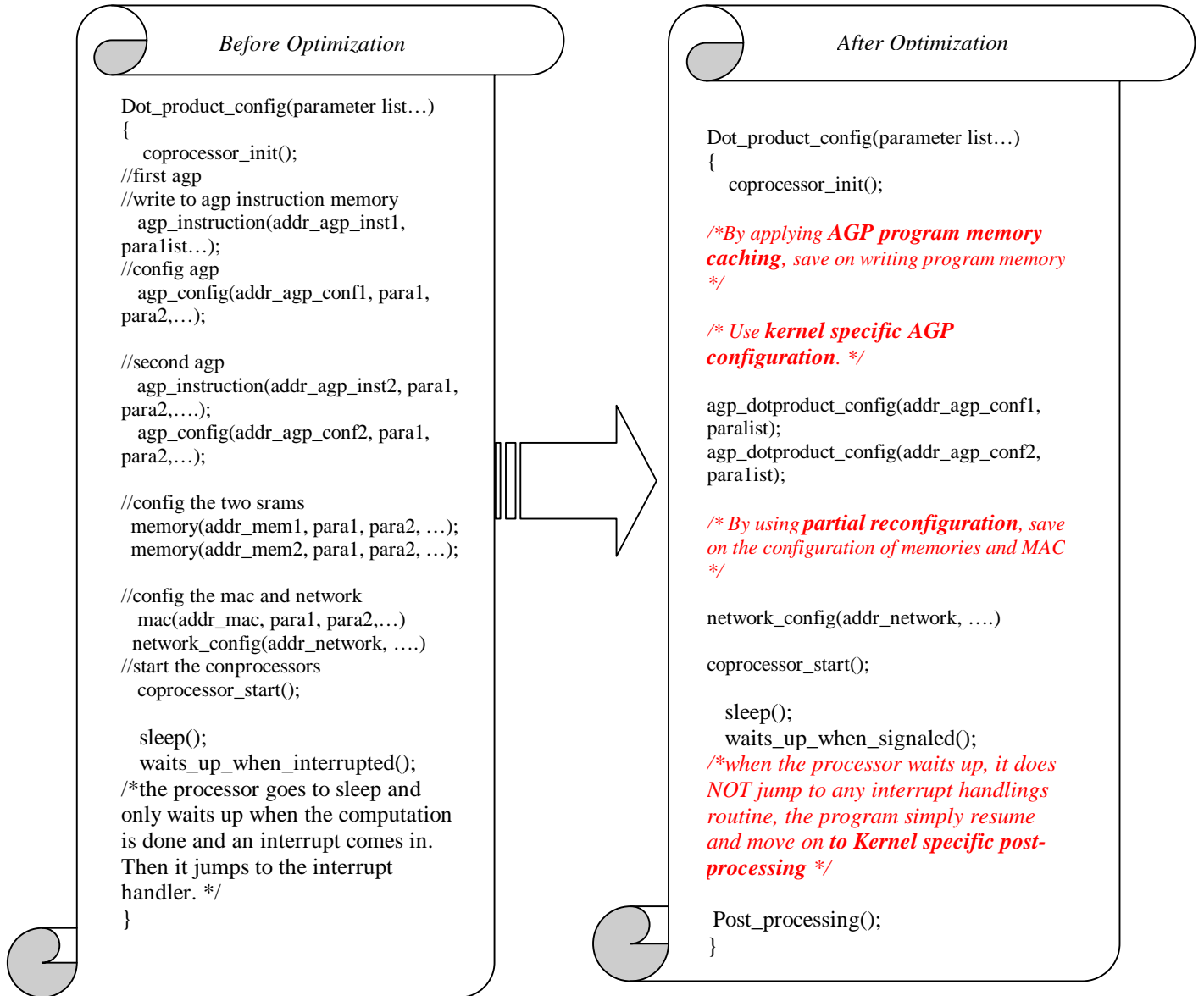


Figure 5. Example kernel program

#### IV. RESULTS

In this section, we will present the evaluation of our code generation process for a speech coding algorithm implemented on a reconfigurable DSP architecture (with an embedded ARM8 [14] as the microprocessor) called Maia. A block diagram of the Maia architecture, which targets baseband source and channel coding algorithms, is shown in the following Figure. The application is the 16-bit

VSELP encoder [15]. The total number of cycles required by VSELP is 126M while it runs entirely on ARM8. After All kernels in the VSELP algorithm have been selected and mapped to the satellite [10], 18.9M cycles remain on the microprocessor (not including configuration cycles).

### Before optimization

This section shows the result before any optimization is applied. Table 2 gives the configuration cycles required to perform all kernels on the satellites. The total cycle count (48.6M) is significantly smaller than the original one but the fact that more time is spent on configuration than computation is not very satisfactory.

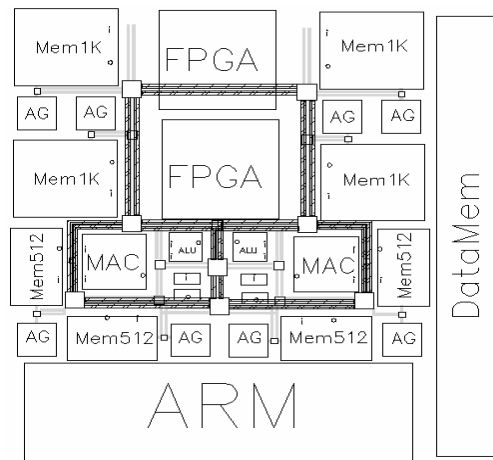


Figure 6 the Maia Architecture

Kernels	Total # of cycle per kernel	# invocation	Subtotal # of configuration cycles in VSELP
FIR	365	4584	1663992
VSMUL	294	2358	688536
IIR	456	250	113500
Dot_Product1 (1 Mac, 1 AGP)	228	7880	1780880
Dot_Product2 (2 Mac, 2 AGP)	302	83802	25140600
Compute_Code	365	997	361911
<b>TOTAL</b>			<b>29749419</b>

Table 2: TOTAL VSELP cycle breakdown per kernel *before* optimization.

Column 2 of Table 2 shows the cycle count per kernel, column 3 shows the number of times each kernel is called in VSELP, column 4 is the product of column 2 and 3.

### Optimization process

We then proceed to apply the local and global optimizations to improve the configuration time. The following table shows the performance gain (% decrease in configuration cycles) in individual kernels by applying a series of optimization techniques.

<i>Kernels</i>	<i>% saving by applying Kernel specific AGP configuration</i>	<i>% saving by applying Kernel specific post processing</i>	<i>% saving by applying AGP program caching and partial reconfiguration</i>
FIR	32.33%	15.07%	9.86%
VSMUL	29.93%	18.71%	8.16%
IIR	32.46%	12.06%	10.53%
Dot_Product1 (1 Mac, 1 AGP)	19.30%	25.88%	5.26%
Dot_Product2 (2 Mac, 2 AGP)	29.14%	19.54%	7.95%
Compute Code	22.74%	15.07%	9.86%

Table 3: Performance saving in individual kernels by applying a series of optimization technique.

Column 2 shows the % decrease in configuration cycles for each kernel type by utilizing Kernel specific AGP configuration. Column 3 shows the % decrease by applying Kernel specific post processing. Column 4 shows the percent decrease by using AGP program caching.

The following graph illustrates the decrease of number of cycles spent on reconfiguring each kernel type in VSELP with each optimization technique. It also shows the overall decrease of the number of reconfiguration cycles in the VSELP application. We can see that the number goes from 29749419 cycles before optimization to 10818115 cycles after optimizations --- a threefold reduction in reconfiguration cycles.

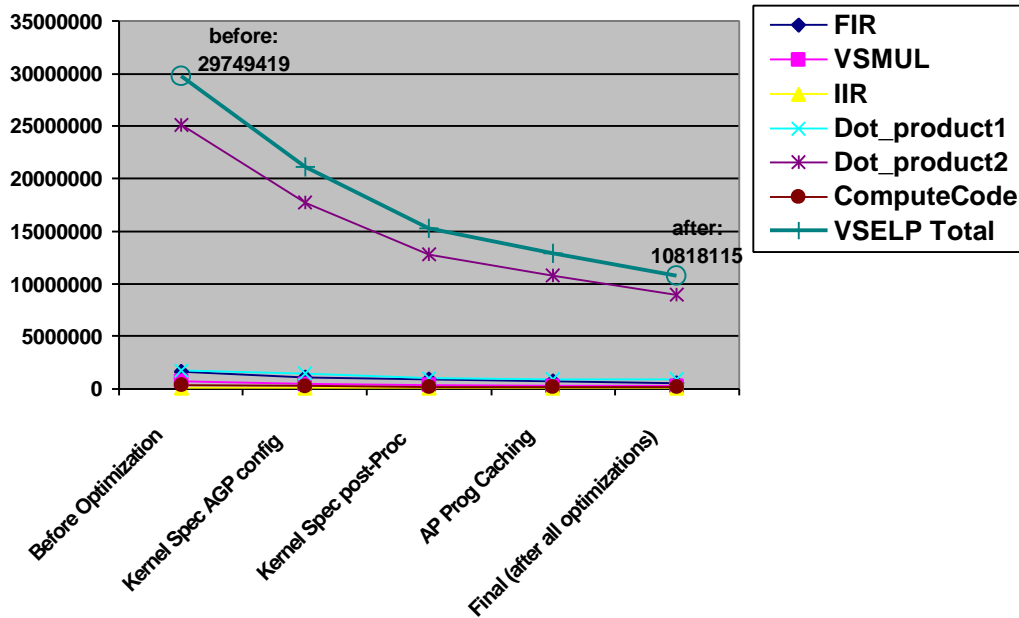


Figure 7. The reduction of reconfiguration cycles with the application of optimization technique

## V. CONCLUSION AND DISCUSSION

In this paper, we presented a code generation process and a set of optimization techniques for reconfigurable architectures. As demonstrated in the paper, a naïve approach could easily generate code that have a high memory demand and is inefficient. In such a case, the overhead of reconfiguring will become a bottleneck of the system. Hence, the code generation process is a very important part of the reconfigurable system design. By carefully considering different system tradeoffs and applying a set of local and global optimization techniques, we are able to improve the performance by a factor of three and greatly reduce the memory requirement.

At present, we addressed the software solution to the code generation process. While one can always write better code, we believe that approaching the problem from the hardware side could push the system performance to a level which software alone can not achieve. Therefore, in our future work, we will also look at the hardware improvement for reconfiguration. Furthermore, we have not considered the overlapping of the ARM processor and the coprocessors. This is due to the sequential nature of our case study application. But in the future, parallelism will be explored to achieve the optimal system performance.

## Acknowledgements

The authors would like to acknowledge DARPA's support of the Pleiades project ( DABT-63-96-C-0026). We also would like to acknowledge the Pleiades Maia design team.

## References

- [1] G. R. Goslin, "A Guide to Using Field Programmable Gate Arrays for Application Specific Digital Signal Processing Performance", Proceedings of SPIE, vol. 2914, p321-331.
- [2] A. Abnous et al, "Evaluation of a Low-Power Reconfigurable DSP Architecture", Proceedings of the Reconfigurable Architecture Workshop, Orlando, Florida, USA, March 1998.
- [3] M. Goel and N. R. Shanbhag, "Low-Power Reconfigurable Signal Processing via Dynamic Algorithm Transformations (DAT)", *Proceedings of Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, CA, November, 1998.
- [4] T. Garverick et al, NAPA1000, <http://www.national.com/appinfo/milaero/napa1000>
- [5] R. Razdan, K. Brace, M. D Smith, "PRISC software acceleration techniques", Proceedings 1994 IEEE International Conference on Computer Design: VLSI in Computers and Processors, Cambridge, MA, USA, Oct. 1994
- [6] J. Hauser and J. Wawrzynnek. GARP: A MIPS processor with a reconfigurable coprocessor. In J. Arnold and K. L. Pocek, editors, Proceedings of IEEE Workshop on FPGA for Custom Computing Machines, Napa, CA, April 1997.
- [7] A. Abnous and J. Rabaey, "Ultra-Low-Power Domain-Specific Multimedia Processors", Proceedings of the IEEE VLSI Signal Processing Workshop, San Francisco, California, USA, October 1996
- [8] S. Hauck, "Configuration Prefetch for single context reconfigurable coprocessors", Proceedings of 1998 International Symposium on Field Programmable Gate Arrays, Monterey, CA, USA, 22-25 Feb. 1998
- [9] Pleiades web page, [http://bwrc.eecs.berkeley.edu/Research/Configurable\\_Architectures/configurable\\_architectures.htm](http://bwrc.eecs.berkeley.edu/Research/Configurable_Architectures/configurable_architectures.htm)
- [10] M. Wan, D. Lidsky, Y. Ichikawa, J. Rabaey, "An Energy Conscious Methodology for Early Design Exploration of Heterogeneous DSPs", Proceedings of the Custom Integrated Circuit Conference, Santa Clara, CA, USA, May 1998.
- [11] H. Zhang, M. Wan, V. George, J. Rabaey, "Interconnect Architecture Exploration for Low Energy Reconfigurable Single-Chip DSPs" Proceedings of the WVLSI , Orlando, FL, USA, April 1999.
- [12] Sanders, a Lockheed Martin Company, "The Design and Implementation of a Context Switching FPGA", Proceedings of FCCM 1998.
- [13] Stanford Unified Intermediate Form, <http://www-suif.stanford.edu/suif/>
- [14] Advanced RISC Machines, ARMulator version 2.10
- [15] I.Gerson and M. Jasiuk, "Vector Sum Excited Linear Prediction (VSELP) Speech Coding at 8Kbps, Proceedings of the International Conference on Acoustic, Speech, and Signal Processing, pp. 461-464, April 1990.