Compiler Optimizations for Adaptive EPIC Processors

Krishna V. Palem[1], Surendranath Talla[1**], and Weng-Fai Wong[2]

[1] Center for Research on Embedded Systems and Technology,
Georgia Institute of Technology
palem@ece.gatech.edu

[2] Dept. of Computer Science,
National University of Singapore

Abstract. Advances in VLSI technology have lead to a tremendous increase in the density and number of devices that can be manufactured in a single microchip. One of the interesting ways in which this silicon may be used is to leave portions of it uncommitted and re-programmable depending on an applications needs. In an earlier paper, we proposed a machine architecture for achieving this reconfigurability and compilation issues that such an architecture will face. In this paper, we will elaborate on the compiler optimization issues involved. In particular, we will outline a framework for code partitioning, instruction synthesis, configuration selection, resource allocation, and instruction scheduling. Partitioning is the problem of identifying code sections that may benefit by mapping them on to the programmable logic resources. The instruction synthesis phase generates suitable implementations for the candidates partitions and updates the machine description database with the new instructions. Configuration selection is the problem of narrowing down the choices of which synthesized instruction (from the set generated by the instruction synthesis phase) to use for each of the code regions that will be mapped to programmable logic. Unlike traditional optimizing compilers, the adaptive EPIC compiler must deal with the existence of synthesized instructions. Compilation techniques addressing each of these problems will be presented.

1 Introduction

Phenomenal advances in semiconductor technology have made it possible to put an increasing amount of silicon devices into the same surface area. This dramatic increase in device density has brought up a fundamental question: how can the extra silicon be effective employed to improve the execution performance of applications? Many straightforward ideas like increasing the number of functional units or the size of the caches run into the problem of diminishing returns.

[**] Author's current affiliation: StarCore Technology Center, Atlanta, GA 30328.

One interesting proposal is to keep a portion of the silicon uncommitted—as re-programmable logic. Processors demonstrating this design, having a core RISC processor and an on-chip, tightly coupled re-programmable logic pool, have been introduced[27]. Depending on the granularity of the re-programmable logic portion, these processors can be used in two ways. The first approach that was adopted by the early generation of such processors is to go for mapping larger granularity computations on the re-programmable logic. The re-programmable logic essentially implements a significantly large grain computation and interfaces it to the application executing in the core is of the nature of a subroutine call. Programmers are expected to know the syntax and semantics of such subroutines and to write the subroutine calls into their application. The alternative that is generally still under-researched is to use the re-programmable logic to implement application-specific instructions. This finer grain approach requires a greater involvement of the compiler. This approach is interesting in that it is an automated approach. On the flip side, we will need a different variation of the generic RISC-core with re-programmable logic processor to support fine grain operations as well as new compiler algorithms.

In this paper, we will describe our proposal for an *Adaptive Explicitly Parallel Instruction Computer* (AEPIC) processor. The bulk of the paper, however, will be devoted to new compiler algorithms needed to generate code for such a machine.

2 Previous Work

The earliest known computing system based on reconfigurable devices wa proposed and implemented by Gerald Estrin at UCLA [11]. It is a hybrid machine consisting of a general purpose processor augmented with high speed logic devices (ALU's, memories) which were interconnected via application specific interconnect. Due to a lack of enabling technology, the reconfiguration was done manually. Mario Schaffner's Circulating Page Structure (CPS) machine [23] implemented a form of hardware paging scheme where the application task was partitioned into pages which circulate through the programmable hardware to compute the task.

The introduction of *field programmable gate array* (FPGA) devices by Xilinx in the mid 80's [32, 26] spurred a flurry of research in the development of FPGA based reconfigurable computing engines. PRISM [2] developed at Brown University demonstrates substantial speedup in the case of large binary operations. PAM, a universal reconfigurable hardware co-processor developed by researchers at DEC Paris Research Labs [3, 4, 29], has been used to demonstrate superior performance/cost ratio compared to every other existing technology of its time on a dozen applications ranging from computer arithmetic, cryptography, image analysis, neural networks, video compression, high-energy physics, biology and astronomy. Another such reconfigurable co-processor board developed by Super Computing Research Center at Maryland called SPLASH-2 [12] has been used to achieve two orders of magnitude speedup on genome sequence matching compared to supercomputers of that time (Cray2). The cover story of an issue of Scientific American [28] written by researchers at UCLA outlines some novel applications of reconfigurable devices.

Other notable reconfigurable computing projects include the Programmable Reduced Instruction Set Computer (PRISC) [21, 22], GARP [16], DISC [31], RAPID [13], the CMU Cached Virtual Hardware (CVH), PipeRench [5], and Chimaera [14, 15].

Reconfigurable Architecture Workstation (RAW) proposed by Agarwal and his colleagues at MIT [30, 19, 1] consists of processors that are sets of interconnected tiles each of which contains instruction and data memories, an arithmetic-logic unit, registers, configurable logic, and a programmable switch that supports both dynamic and static (compiler orchestrated) routing, interconnected by programmable interconnects. Compiling to a RAW machine is complicated by two factors:

– Unlike traditional super-scalars, a RAW processor does not bind specialized logic structures such as register renaming or dynamic instruction issue logic into hardware. Scheduling and resource allocation are the responsibility of the compiler.
– Communication patterns within the code need to be analyzed in order to schedule inter-tile communication.

Preliminary experience with compiling to the RAW machine can be found in [1, 19].

So far, most research efforts have focused on the architectural aspects of reconfigurable systems. Little attention has been paid to compilation issues. Many performance studies have been done and impressive speedups were demonstrated. However, important issues like compilation times, target cost have been neglected. Our research attempts to address these latter issues.

3 Adaptive Explicitly Parallel Instruction Computing (AEPIC)

Processors that take advantage of instruction level parallelism generally fall into two main categories: superscalar (dynamically scheduled) and VLIW (statically scheduled). With the addition of features such as predicated execution, support for software pipelining etc., the latter has been renamed Explicitly Parallel Instruction Computer (EPIC). We refer to Schlansker and Rau [24] for an evaluation of these two ILP approaches. It is to EPIC that we propose adding a reconfigurable component that is amenable to compiler optimizations. We call this new configuration Adaptive EPIC (AEPIC) [25]. Figure 1 shows the abstract execution model for AEPIC. The “hardwired component” in Figure 1 is the EPIC core processor. The adaptive component consists of functional units that have been configured into the datapath by some reconfiguration instructions. Operations performed by the configured functional units are triggered by specific AEPIC instructions invoked on the hardwired functional units.

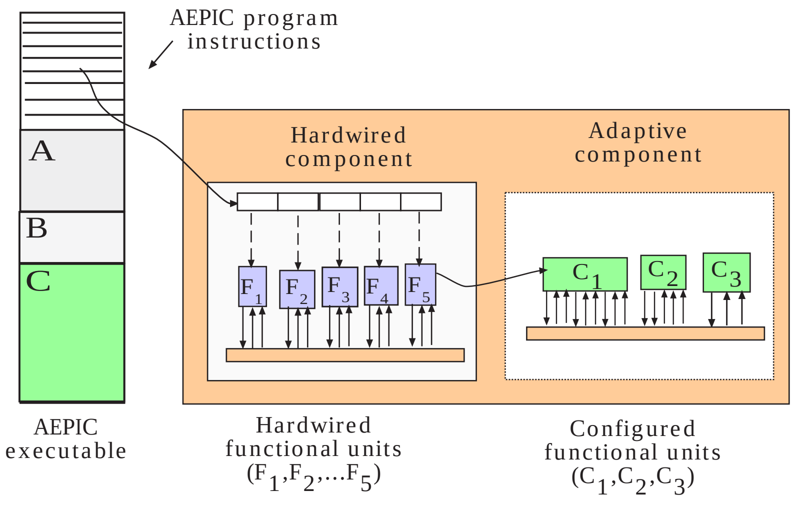

AEPIC program instructions
A
B
C
Hardwired component
Adaptive component
F1
F2
F3
F4
F5
C1
C2
C3
AEPIC executable
Hardwired functional units (F1,F2,...F5)
Configured functional units (C1,C2,C3)

Fig. 1. AEPIC executable and abstract data-path

Figure 2 shows the details of a AEPIC machine. The core component consists of a standard EPIC machine. The adaptive component of the AEPIC processor consists of the *Configuration Cache Hierarchy*, *Multi- context Reconfigurable Logic Array* (MRLA) and *Array Register File* (ARF) connected together via bus interconnect. The MRLA provides the programmable logic resources to host the *Configured Functional Units* (CFUs). The *C-Cache* serves as a temporary cache for configurations before they are instantiated on the MRLA. This is analogous to the way registers serve as storage for program values. The rest of the configuration cache hierarchy consists of the C1 cache connected to external memory. The *Configuration Register File* (CRF) consists of a set of *configuration registers* (CRs). Each CR serves as an alias to either a configured functional unit or a configuration allocated in the C-Cache. Most of the AEPIC instructions take a configuration register as an operand. These are the AEPIC instructions that perform operations such as delete a CFU, etc. The instruction refers to the CFU by its alias—the configuration register. For example, the delete CFU operation in AEPIC is `DELC cr, L, p`. Here, `cr` is the configuration register associated with the desired CFU, `L` is the latency assumed by the compiler for this operation and p the predicate guard for this operation.

We shall now give a more detailed description of the MRLA as it is pertinent to the compiler optimizations discussed in the rest of the paper. The Multi-context Reconfigurable Logic Array (MRLA) is the primary resource used for hosting the configured functional units. Like a typical Field Programmable Gate Array (FPGA), the MRLA is a two dimensional region of the processor die that

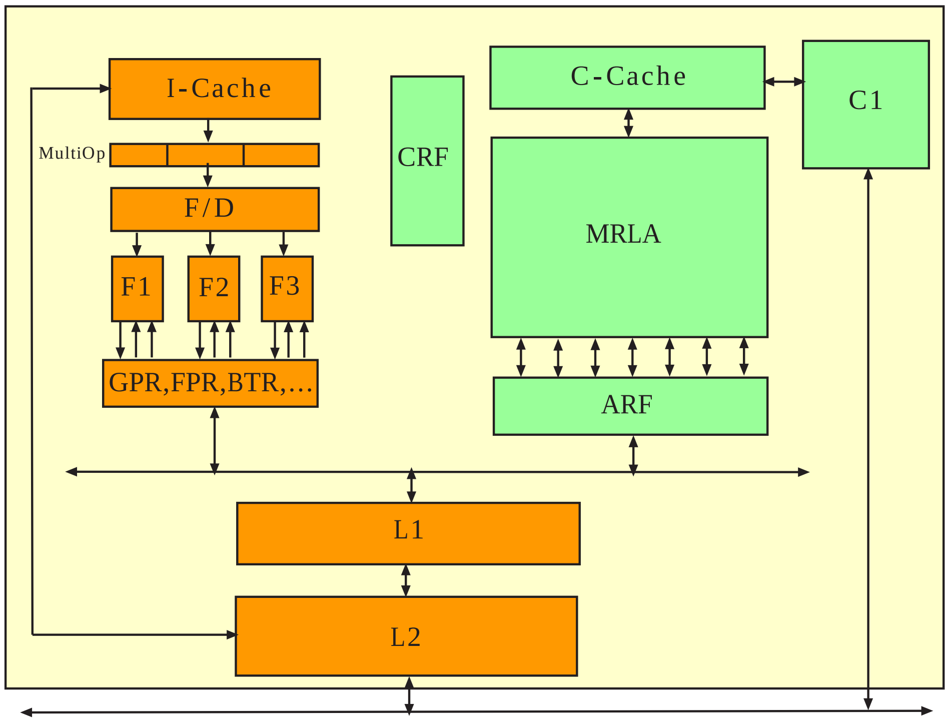
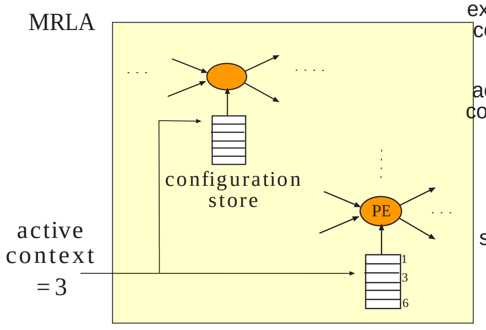


Fig. 2. AEPIC machine model

is composed of programmable logic and interconnect blocks. We shall use the term *Programmable Element* (PE) to refer to both the programmable logic block as well as the programmable interconnect block. Each PE is associated with a *configuration instruction* (its "program") which determines the behavior of that programmable element. Any given logic design can be emulated on the MRLA by supplying suitable *configuration instructions* for all the programmable elements of the array. In the context of AEPIC machines, these logic designs are the CFUs.

In a standard FPGA only one configuration instruction is associated with each programmable element. This implies that only one logic design can be resident on the array until the device is reconfigured (i.e., a new set of configuration instructions are associated with the programmable elements of the array). In an MRLA, each programmable element can be associated with multiple configuration instructions. This allows multiple logic designs (CFUs) to be simultaneously resident on the MRLA. The desired logic design can be activated by selecting the appropriate configuration instruction for each of the programmable element.

Configuration instruction slots for each PE (called the *configuration memory*) are stored in an ordered sequence and all PEs have the same number (D) of configuration instruction slots. MRLA takes an input called *context_id* which can take values from 1 to D. A value of k to the *context_id* input selects the



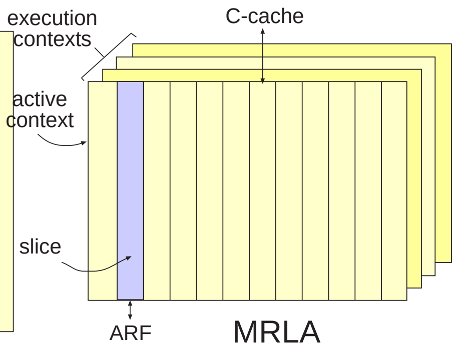


Fig. 3. Structure of MRLA

Fig. 4. MRLA multiple contexts

k^{th} configuration instruction from the configuration memory as the instruction for each PE. The k^{th} confi guration instruction is referred to as the **active configuration instruction** for that PE.

The set of configuration instructions with identical index in the configuration memory of a PE is referred to as an **execution context**. The execution context that is associated with currently active configuration is called the **active context**. MRLA can be effectively viewed as an array of FPGAs, one array per execution context; and the *context_id* serves as the index into this array. Selection of an execution context, makes all the CFUs of that context available for instruction processing by subsequent instructions.

With this brief introduction to our proposed AEPIC architecture, we shall now deal with the compilation issues of such a machine.

4 Compilation Framework

Inputs to an AEPIC compiler are (a) the application source program written in a high level language such as C/C++, (b) a description of the particular instance of the AEPIC processor described in a machine description language (c) a library containing parameterized configurations for popular computational routines such as FFT, DCT, etc. The input source code may be instrumented with pragmas intended to give partitioning/mapping hints to the compiler. These cues are communicated to different phases of the compilation process through the intermediate code representations.

The first phase is a standard lexical and syntactic analysis phase. The partitioning module will help delineate portions of the program that might benefit from execution on the configurable portion of the target. The partitioning phase is followed by two independent phases that may be performed in parallel: the high-level optimization phase and the operation synthesis phase. In the high-level optimization phase, the code to be executed on the hardwired functional units (which execute instructions from the AEPIC ISA) is optimized as is typically performed in a standard ILP compiler. The instruction synthesis phase generates suitable mappings of the code partitions identified by the partitioning phase. Each of these mappings are packaged as *custom operations*. The machine description is updated with the synthesized instructions. The instruction synthesis phase may generate multiple implementations for the identified partitions reflecting different performance/device-area tradeoffs. The configuration selection phase tags different regions of the intermediate code with semantically equivalent custom operations. At the end of the configuration selection phase, every region of the code identified for mapping on to the configurable part of the target has a unique configuration associated with it.

Subsequent phases of the compilation are similar in structure to back-end phases of a typical ILP compiler suitably adapted to consider the special characteristics of configurations/CFUs. A typical ILP compiler back-end comprises of (at the minimum) the following phases, in sequence: pre-pass scheduling, register allocation, post-pass scheduling and code generation. In addition to these, AEPIC compilation introduces an extra phase: *configuration allocation*. The configuration allocation phase is aimed at optimizing allocation of resources for configurations—a task analogous to that of register allocation. The resources meant for configuration are the C-cache and the MRLA. These resources are independent of those intended for register allocation. Hence, configuration allocation may be performed in parallel or in any order with respect to register allocation. The main task of the scheduler is to reduce the critical path through the code by masking reconfiguration overheads.

In the above framework, the scheduled and allocated code is transformed into machine code, which is then translated to object code for simulation. Simulation yields correctness as well as performance data for the program on the given input data. The execution profile can be re-instrumented into the IR for profile based optimizations.

4.1 Partitioning

Partitioning is the task of determining the set of code segments (referred to as *candidates*) in the application which may be synthesized as application specific instructions. These application specific instructions are implemented on the MRLA as configured functional units.

The partitioning module takes as input (a) an intermediate code representation of the source program and (b) the machine description of the target AEPIC processor and identifies regions of the intermediate code that can benefit from mapping to the MRLA. Although not necessary, the partitioner can only benefit from an execution profile of the application. The execution profile gives execution frequencies of different regions of the intermediate code. This information can be obtained by compiling to a base EPIC architecture and re-instrumenting the intermediate code with the profile data from the simulator. Since the reconfiguration overhead can be quite large, it may not pay to reconfigure the processor for a certain segment of the code if it is known that this section will rarely be executed. The execution profile can be used by the partitioner to eliminate such code segments from consideration for mapping onto the MRLA.

The code-partitioner is composed of four main steps. These steps are performed in sequence on each intermediate representation that is created during the compilation process.

1. Profile. The profile phase is composed of four steps:

- *Instrument:* The IR nodes are tagged with code for gathering the execution profile for each node. So for example, in a basic block IR, each basic block will be associated with a "execution frequency" variable and a piece of code at the entry of the basic block which updates this variable whenever control enters the basic block.
- *Translate:* During this step the instrumented IR is translated to C language code which is then compiled using the native compiler to yield a semantically equivalent (to the application program) executable with one additional attribute: this executable also gathers the runtime properties of the program as specified by the instrumentation code.
- *Execute:* The execute step runs the instrumented and compiled application using the inputs associated with the application program.
- *Re-instrument:* After the execute step, the application generates the execution profile (written out by the instrumented code) and re-instruments the starting IR with the actual execution profile values (such as the frequency of execution of a basic block, etc.)

2. Analyze. After the **profile** step, the IR is tagged with the execution profile. This information is used to identify the most frequently executing regions of the code. The **analyze** phase traverses the IR and determines various properties of the program that may be useful in determining if a region is a candidate for partitioning. Some of these properties are bit-widths/types of variables, whether certain arrays are constants or are bounded of known dimensions, etc.

3. Identify. The **identify** step performs a bottom-up traversal of the IR and tags each region as a candidate for mapping using certain heuristics based on control structure, operation types and other attributes identified in the **analyze** phase. The identify phase also considers machine resource constraints (from the machine description database) and any user supplied cues transmitted through the IR, in deciding whether a region of the IR is a feasible candidate for mapping.

4. Coalesce. The coalesce phase merges adjacent regions marked by the **identify** phase if the merged regions can still be accommodated on the MRLA assuming the given machine constraints.

4.2 Instruction Synthesis

Instruction synthesis is a systematic technique for defining new instruction set for a given micro-architecture. The process typically involves analyzing the benchmark(s) to infer the most suitable operation repertoire based on its computational characteristics for the intended micro-architecture. In certain cases the micro-architecture itself is synthesized in parallel with the instruction set.

The Instruction Synthesis (IS) module takes a list of candidate partitions that have been identified for mapping onto the programmable logic and synthesizes a set of functional units that can implement all the computations of the code partitions. In addition to the candidate partitions list, the IS module may also take as input, a library of pre-synthesized macros for various basic operators and frequently used kernels such as FFT, DCT, FIR/IIR filters, etc. The purpose of this library is to speedup the process of mapping a given partition to the MRLA. It helps to keep these pre-synthesized macros generic so that they are applicable to a wide range of the target programmable logic parameters, and also accommodate variations in the structure of the input partitions.

4.3 Configuration Selection

After the partitioning and instruction synthesis phase, all the regions of the intermediate code that may be mapped to the MRLA have been identified. Each of the candidate partitions can now be replaced with one of the equivalent configurations synthesized by the IS phase. Configuration Selection (CS) is the problem of determining which semantically equivalent synthesized configuration (custom instruction) to associate with each candidate partition that is mapped to MRLA.

We extend the EPIC code generation path to include configuration selection. The key difference compared to traditional instruction selection is that the selection can happen at multiple levels in the IR. Configuration selection can happen for leaf level operations as well as higher level structures such as groups of instructions, program statements, loops, and functions, etc.

4.4 Configuration Allocation

After configuration selection, some of the nodes of the intermediate code may be tagged with application specific instructions. The configuration selection module *does not consider availability of resources on chip* when it decides on which configuration to assign with each partition that is synthesized into application specific instruction. Any realistic AEPIC processor would have limited programmable logic resources available for CFUs and hence it is quite possible that all the desired configurations cannot be accommodated on chip simultaneously.

Configuration Allocation (CA) is the problem of optimally allocating/de-allocating on-chip resources that are intended for holding configurations (on C-cache) or CFUs (on MRLA). Poor allocation of resources for configurations can lead to long periods of processor stalls caused either due to waiting for the configurations to load into the MRLA or due to excessive thrashing in the configuration memory hierarchy. Hence, optimal allocation of configurations to C-cache and MRLA resources is critical for achieving high performance on an AEPIC processor. In addition, we would like the allocation algorithm itself to be time and space efficient.

This problem is analogous to the problem of implementing virtual memory on systems with limited physical memory or to the problem of allocating registers for program variables performed in most standard compilers. In the remainder of this section, we present a formal definition of the configuration allocation problem and relate it to the well studied problem of register allocation. We show how extensions to the register allocation techniques yield solutions to the configuration allocation problem.

Register Allocation (RA) allocates program variables (also referred to as temporaries) to registers in order to minimize the overall number of accesses to external memory. Fundamentally, both register allocator and configuration allocator perform the same task—allocation of on-chip resources for program values (variables in the case of RA and configurations in the case of CA). Configuration allocation differs from register allocation in the following ways:

1. *Non-uniform allocation units.* Sizes of configurations are typically much larger and *vary widely* compared to the sizes of data values stored in registers which are much smaller and almost always uniform in size.
2. *Heterogeneous resources to be allocated.* There are two types of local memories for configurations: (1) the C-cache and (2) the MRLA. These resources differ in their capacities, access (read/write) costs and sizes of allocation units. There is only one type of resource to be allocated in register allocation - the register set.
3. *Immutable values.* Currently, AEPIC architecture configurations are immutable. So memory write back of configurations is not an issue.
4. *No copies or moves.* AEPIC does not provide any architecturally visible features to create copies or move configurations with in the MRLA or the C-cache.

There are two types of storage classes are available for hosting configurations: (1) C-cache and (2) MRLA. Allocated configurations are present in exactly one of these storage classes. Every allocated configuration whether it is on the MRLA or in the C-cache is associated with a distinct configuration register from the configuration register file (CRF). The unit of allocation in the C-cache is a C-cache *block* and on the MRLA it is a *slice*. All configurations are constrained to consume an integral number of consecutive slices in the MRLA. Configuration data for each MRLA slice requires an integral number of blocks in the C-cache. The basic steps involved in using configurations are:

1. Allocate a configuration register with the configuration to be loaded.
2. Allocate requisite number of blocks in the C-cache for the configuration. If the C-cache is insufficient for the configuration, then the allocation fails and application is aborted. If the configuration is smaller than the C-cache size but the total free space is less than that which the configuration requires, then some of the resident configurations are evicted (since configurations are immutable, they are simply deleted and not written back to memory).
3. Schedule the loading of configuration data into the C-cache into the allocated blocks.
4. Allocate consecutive slices on the MRLA to load the configuration that was loaded into the C-cache (to instantiate the CFU corresponding to the configuration).

5. Schedule transfer of configuration data from the C-cache to MRLA.
6. One or more operations are executed on the CFU.
7. At some point if the MRLA resources are required for another CFU or if this configuration will not be used any more, then it is evicted to the C-cache (if it may be used again) or just deleted from MRLA.
8. Allocated configuration register is freed - it can be allocated to a new configuration.

Note that the same configuration register refers to the configuration data when it was in the C-cache and to the corresponding CFU when loaded onto the MRLA. Once the configuration is completely moved to MRLA from the C-cache, the C-cache resources allocated for the configuration may freed since it is wasteful blocking those resources as long as the configuration data is available on the MRLA. However, in certain cases it might be useful to architecturally expose the deallocation operation. For example, if multiple instances of the CFU corresponding to the configuration are needed on the MRLA, it might be more efficient to copy the configuration data from the C-cache to the MRLA once for each of the CFU instances instead of loading from external memory into the MRLA to make copies of the CFU already available on the MRLA.

4.5 A Simplified AEPIC Allocation Model

For the remainder of this section we consider a simplified version of the configuration allocation problem. In the simplified version, the machine is assumed to contain N configuration registers and each "virtual" configuration in the program intermediate code specifies an integer k which is the number of physical configuration registers it requires. The only difference between this problem and the conventional register allocation is that, in the conventional register allocation problem, each virtual register (also called program temporary) is assigned to a single physical register.

A live range is an isolated and connected group of nodes in the control flow graph that connects the definitions and uses of a given program variable. It is the principle data structure for register allocation. Two live ranges interfere if one of them is live at the definition point of the other. As the reader may have already noted, there is a close resemblance between register live ranges and configuration. The algorithm to construct the interference graph for configurations is very similar to that for register live ranges.

Pruning For Configuration Allocation If there are program regions where the total resource requirement of configurations that are live at that point exceeds the available resources on the processor, configuration allocation will fail. Analogous to the concept of *register pressure*, we define the total resource requirement at any program point the *resource pressure* at that point. *Pruning* is a technique of preprocessing the live ranges of configurations to ensure that the *resource pressure* never exceeds the total resources available on the machine.

Pruning involves (a) determining the set of live ranges to split and, (b) determining the right split points for the selected live ranges. Once a live range is split, compensation code needs to be inserted to fetch the values (configurations) for the uses encountered in the second (or later, if multiple splits were performed) part of the live range. One downside of reducing the resource pressure is the additional time taken for executing the compensation code. Hence pruning decisions cannot be made arbitrarily. In addition to the resource pressure, execution frequency should be taken into account before a selecting a live range for splitting since that determines the number of times the compensation code would be executed.

We propose a pruning technique based on the hierarchical technique proposed by Callahan and Koblenz [6]. In their algorithm, gaps between references to a register in a live range are identified. These are possible regions which can be spilled to memory. The maximal length live-range gap is referred to as *wedges* in [9]. Non-overlapping and maximal wedges are identified using the control tree [20]. The choice of wedge s to prune is a function of (a) the runtime cost of compensation code that would be added in the pruned region and, (b) size of the region of the program region that would benefit from the pruning decision. Our algorithm takes into account the special requirements of configurations (their non-uniform sizes - which implies non-uniform spill costs; immutability - which implies stores to memory are not required) and also enhances the scheme with regard to identifying spill candidates and spill locations based on execution profile as well. The algorithm, adapted from the register live range pruning algorithm from [9] and is described in Algorithm 1. The relevant parameters and the data-structures used by the algorithm are listed below.

$$
\begin{aligned}
R &= \text{total number of recourse units available for allocation}\\
C &= \text{configurations (candidates) to be allocated}\\
T &= \text{Control tree of the program}\\
ResUnits[c] &= \text{number of resource units required by the configuration } c\\
Live[n] &= \text{set of configurations live in control node } n\\
Refs[n] &= \text{configurations that are referenced in control node } n\\
Wedges[n] &= \text{list of candidates with wedges that have heads at } n\\
Freq[e] &= \text{number of times control traversed along edge } e\\
Excess[n] &= \begin{cases} \sum_{c \in Live[n]} ResUnits[c] - R & n \in T.Leaves,\\ Max_{c \in T.Children[n]}\{Excess[c]\} & n \notin T.Leaves. \end{cases}\\
LiveUnits[n, C] &= \begin{cases} 1 & n \in T.Leaves \wedge C \in Live[n],\\ 0 & n \in T.Leaves \wedge C \notin Live[n],\\ \sum_{p \in T.Children[n]} LiveUnits[p, C] & n \notin T.Leaves. \end{cases}
\end{aligned}
$$

4.6 Graph Multi-coloring Configuration Allocator (GMCA)

Here we provide a simple configuration allocator called Graph Multi-coloring Configuration Allocator (GMCA), based on Chaitin's graph coloring register allocator [7, 8] wi th the spill and split decision modifications suggested by Hansoo and Leung [18] for the simplified AEPIC configuration resource model. All past graph coloring based register allocation schemes are based on the idea of *simplification* [17]. If a graph G contains a node v with fewer than K neighbors and if $G - v$ can be colored with K colors, then G is K colorable. In the case of *configuration allocation*, multiple colors may be allocated to each node and hence it calls for a stronger version of the simplification step. We first state and prove this *simplification lemma* and then show it is used in our GMCA algorithm.

Let $G(V, E, w)$ be an undirected graph where $w : v \to \mathbf{Z}$ is a weight function defined on the vertices. Let $C : v \to S$ be a function on the vertex set such that $S \subset \{1, \ldots, K\}$. Then C is a valid *K-multi-coloring* of the graph if $|C(v)| = w(v)$ and $\forall e \in E, where\ \ e = (u, v),\ C(u) \cap C(v) = \emptyset$.

Lemma 1. [Simplification lemma]
Let vertex $v \in V$ be such that $\sum_{u \in Adj(v)} w(u) \leq K - w(v)$. Let $G' = G - v$ be the subgraph obtained by removing v and its incident edges from G. If G' can be K-multi-colored, then so can G.

Proof: Let C be the K-multi-coloring of G'. Let $S_v = \bigcup_{u \in Adj(v)} C(u)$. By definition of C, $|C(u)| = w(u)$. This implies that $|S_v| \leq \sum_{u \in Adj(v)} w(u)$. Given that $\sum_{u \in Adj(v)} w(u) \leq K - w(v)$, it implies that $|S_v| \leq K - w(v)$. Consider $C_v = \{r | r \in \{1, \ldots, K\}, r \notin C(u)\}\}$. Clearly, $|C_v| \geq K - w(v)$. Let $C_v^{w(v)}$ be any $w(v)$ sized subset of C_v. Let

$$C'(p) = \begin{cases} C(p) & \text{if } p \in V(G') \\ C_v^{w(v)} & \text{if } p = v \end{cases}$$

Then, C' is a valid *K-multi-coloring* of G.

A vertex $v \in V$ such that $\sum_{u \in Adj(v)} w(u) \leq K - w(v)$ then the vertex is called *unconstrained* else it is referred to as an *constrained* node. Pseudo-code for the Graph Multi-coloring Configuration Allocator (GMCA) based on the simplification lemma is presented in Algorithm 2.

Phases of graph multi-coloring configuration allocator.

1. *Build:* Live ranges are computed and the interference graph is constructed during this phase.
2. *Coalesce:* The coalesce step removes any unnecessary move (copy) instructions effectively merging the live ranges corresponding to the values connected by the move instruction.

3. *Simplify:* The *simplification lemma* is the basis for this step. *Unconstrained* nodes are selected and pushed onto the *color_stack* and removed from the interference graph. Since the *simplification lemma* guarantees that unconstrained nodes can always be colored if the reduced graph is colorable, they are removed from the graph. This step in turn might enable other *constrained* nodes to become *unconstrained*. If so, then it is repeated until either the graph is empty or all nodes are *constrained*.
4. *Prioritize:* Priorities are assigned to constrained live ranges. Live ranges are selected for coloring in order of their priority. Priority functions capture the expected benefit of allocating the live range to on chip resources as opposed to external memory.
5. *ProcessNode:* The highest priority node is selected for coloring. For each node, just as in [10], a *Forbidden* set is maintained which indicates the set of colors (resources) that have already been allocated and hence cannot be used for the current node. If there are enough available colors that can satisfy the color requirement for the current node, the node is colored and the *Forbidden* sets of its neighbors are updated to reflect the allocation. If the set of available colors cannot satisfy the demand for this node, then the node is either *spilled* or the live range *split* depending on which one is most beneficial.
6. *ProcessStack:* Unconstrained nodes eliminated (pushed onto the color_stack) during the simplify phase are colored in the reverse order in which they were removed from the graph. The original graph is incrementally reconstructed by adding one node at a time from the top of the *color_stack* and is assigned the color vector. The *simplification lemma* guarantees that enough colors are available to color the inserted node.

5 Conclusion

In this paper, we defined the problems encountered in each of the phases of the compilation path for AEPIC. In some cases we proposed techniques that may be used to solve these compilation problems. Substantial research still needs to be done and we do not make any claims on whether any of these techniques are sufficient to fully exploit the capabilities of AEPIC processors.

References

1. Anant Agarwal, Saman Amarasinghe, Rajeev Barua, Matthew Frank, Walter Lee, Vivek Sarkar, Devabhaktuni Srikrishna, , and Michael Taylor. The RAW compiler project. In *Proceedings of the Second SUIF Compiler Workshop*, pages 21–23, Stanford, CA, August 1997.
2. P. M. Athanas and H. F. Silverman. Processor reconfiguration through instruction-set metamorphosis. *IEEE Computer*, 26(3):11–18, March 1993.
3. P. Bertin, D. Roncin, and J. Vuillemin. Introduction to programmable active memories. In J. McCanny, J. McWhirther, and E. Swartslander Jr., editors, *Systolic Array Processors*, pages 300–309. Prentice Hall, 1989.

4. P. Bertin, D. Roncin, and J. Vuillemin. Programmable active memories: a performance assessment. In G. Borriello and C. Ebeling, editors, *Research on Integrated Systems: Proceedings of the 1993 Symposium*, pages 88–102, 1993.
5. Srihari Cadambi, Jeffrey Weener, Seth Copen Goldstein, Herman Schmit, and Donald E. Thomas. Managing pipeline-reconfigurable fpgas. In *Proceedings ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*, February 1998.
6. D. Callahan and B. Koblenz. Register allocation via hierarchical graph coloring. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, volume 26, pages 192–203, Toronto, Ontario, Canada, June 1991.
7. G. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markstein. Register allocating via coloring, 1981.
8. Gregory J. Chaitin. Register allocation and spilling via graph coloring. *SIGPLAN Notices (Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, Boston, Mass.)*, 17(6):98–105, 1982.
9. F. Chow, K. Knobe, A. Meltzer, R. Morgan, and K. Zadeck. Register allocation.
10. Fred C. Chow and John L. Hennessy. Register allocation by priority–based coloring. In *Proceedings of the ACM SIGPLAN 84 Symposium on Compiler Construction*, pages 222–232, New York, NY, 1984. ACM.
11. G. Estrin. Organization of computer systems – the fixed plus variable structure computer. In *Proceedings of the Western Joint Computer Conference*, pages 33–40, 1960.
12. M. Gokhale, W. Holmes, A. Kopser, S. Lucas, R. Minnich, D. Sweely, and D. Lopresti. Building and using a highly parallel programmable logic array. *IEEE Computer*, 24(1):81–89, January 1991.
13. C. Ebeling D. C. Green and P. Franklin. RaPiD – reconfigurable pipelined datapath. In R. W. Hartenstein and M. Glesner, editors, *Field-Programmable Logic: Smart Applications, New Paradigms, and Compilers. 6th International Workshop on Field-Programmable Logic and Applications*, pages 126–135, Darmstadt, Germany, September 1996. Springer-Verlag.
14. S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao. The chimaera reconfigurable functional unit. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 87–96, 1997.
15. S. Hauck, M. M. Hosler, and T. W. Fry. High-performance carry chains for fpgas. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 223–233, 1998.
16. John R. Hauser and John Wawrzynek. GARP: A MIPS processor with a reconfigurable coprocessor. In J. Arnold and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 12–21, Napa, CA, April 1997.
17. A. Kempe. The geographical problem of the four colors. *Amer. J. Math. 2, 193–200.*, 1879.
18. H. Kim and A. Leung. Frequency based live range splitting. *Technical report, ReaCT-ILP Laboratory, New York University*, 1999.
19. Walter Lee, Rajeev Barua, Devabhaktuni Srikrishna, Jonathan Babb, Vivek Sarkar, Saman Amarasinghe, and Anant Agarwal. Space-time scheduling of instruction-level parallelism on a RAW machine. *MIT/LCS Technical Memo TM-572*, December 1997.
20. S. Muchnick. Advanced compiler design and implementation, 1997.

21. R. Razdan and M. D. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 172–80. IEEE/ACM, November 1994.
22. Rahul Razdan. *PRISC: Programmable Reduced Instruction Set Computers*. PhD thesis, Harvard University, May 1994.
23. Mario R. Schaffner. Processing by data and program blocks. *IEEE Transactions on Computers*, 27(11):1015–1028, November 1978.
24. M. Schlansker and B. Rau. EPIC: An architecture for instruction-level parallel processors. *Technical report HPL-1999-111, Hewlett-Packard Laboratories, Technical Publications Department, 1501 Page Mill Road, Palo Alto, CA 94304.*, 2000.
25. S. Talla. *Adaptive Explicitly Parallel Instruction Computing.* PhD thesis, New York University, 2000.
26. Stephen M. Trimberger. *Field-Programmable Gate Array Technology.* Kluwer Academic Publishers, 1994.
27. Triscend Corp., Mountain View, U.S.A. *Triscend A7 Configurable System-on-a-Chip Family Data Sheet*, 2001.
28. John Villasenor and William H. Mangione-Smith. Configurable computing. *Scientific American*, pages 66–71, June 1997.
29. J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard. Programmable active memories: Reconfigurable systems come of age. *IEEE Transactions on VLSI Systems*, 4(1):56–69, 1996.
30. E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to software: RAW machines. *IEEE Computer*, pages 86–93, September 1997.
31. M. J. Wirthlin and B. L. Hutchings. DISC: The dynamic instruction set computer. In J. Schewel, editor, *Proceedings of the International Society of Optical Engineering (SPIE). Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing.*, volume 2607, pages 92–103, Philadephia, PA, 1995.
32. Xilinx, San Jose, CA. *The Programmable Logic Data Book*, 1994.

Algorithm 1 Pruning

```
function InitPrune(N) {
    if ($N \in T.Leaves$) {
        $LivePart[N] \leftarrow Live[N]$;
        $Excess[N] \leftarrow (\sum_{n \in Live[N]} ResUnits[n]) - R$;
        $\forall C \in Live[N]: \quad LiveSize[C, N] \leftarrow |C|$;
    } else {
        $\forall M \in N.children: \quad InitPrune(M)$;
        $LivePart[N] \leftarrow \cup_{M \in N.children} LivePart[M]$;
        $Refs[N] \leftarrow \cup_{M \in N.children} Refs[M]$;
        $Excess[N] \leftarrow Max_{M \in N.children} Excess[M]$;
        $\forall C \in LivePart[N]: \quad LiveSize[C, N] \leftarrow \sum_{M \in N.children} LiveSize[C, M]$ ;
        $\forall C \in Refs[N] \wedge \forall M \in N.children$:
        if ($C \notin Refs[M] \wedge C \in LivePart[M]$)
            NewWedge(C,M);
    }
}
function UpdatePressure (W,N) {
    if ($N \in T.leaves$) {
        Live[N] $\leftarrow$ Live[N] - {W};
        $Excess[N] \leftarrow Max\{(\sum_{n \in Live[N]} ResUnits[n]) - R, 0\}$;
    } else {
        $Excess[N] \leftarrow Max_{m \in N.Children}\{UpdatePressure(W, m)\}$;
    }
    return $Excess[N]$;
}
function Prune (N) {
    PrioritizeWedges(N);
    while ($Excess[N] > 0 \wedge |Wedges[N]| > 0$) {
        $W \leftarrow Wedges[N].top()$;
        PruneWedge($W$);
        UpdatePressure($W, N$);
    }
    $\forall M \in N.Children \wedge Excess[N] > 0$:    Prune($M$);
}
```

Algorithm 2 Graph multi-coloring configuration allocator

```
function GMCA($CFG\ cfg$) {
    $G \leftarrow Build(cfg)$;
    while ($G \neq \emptyset$) {
        while ($\exists v | v$ is unconstrained) {
            $S.push(v)$;
            $G \leftarrow G - \{v\}$;
        }
        if ($G \neq \emptyset$) {
            $ComputePriorities(G, h)$;
            $v \leftarrow HighestPriorityNode(G)$;
            if ($IsColorable(v)$) {
                Color(v);
            } else {
                $G \leftarrow Split(G, v)$;
            }
        }
    }
    $ProcessStack(S)$;
}
function IsColorable($v$) {
    return ($v.ColorReq \geq (K - \sum_{0<i\leq K} v.Forbidden[i])$);
}
function Color($v$) {
    $availColors \leftarrow \overline{v.Forbidden}$;
    $P \leftarrow Min_{\{i|0<i\leq K\}}\{\sum_{0<k\leq i} availColors[k] = v.ColorReq\}$;
    $mask \leftarrow 1^P 0^{N-P}$;
    $v.Assignment \leftarrow v.availColors \wedge mask$;
    $\forall u | u \in Adj(v): \quad u.Forbidden \leftarrow u.Forbidden \vee v.Assignment$;
}
```