# Automatic Data Partitioning for the Agere Payload Plus Network Processor

Steve Carr
Department of Computer Science
Michigan Technological University
Houghton MI 49931-1295
carr@mtu.edu

Philip Sweany
Department of Computer Science and Engineering
University of North Texas
P.O. Box 311366
Denton TX 76203
sweany@cse.unt.edu

## ABSTRACT

With the ever-increasing pervasiveness of the Internet and its stringent performance requirements, network system designers have begun utilizing specialized chips to increase the performance of network functions. To increase performance, many more advanced functions, such as traffic shaping and policing, are being implemented at the network interface layer to reduce delays that occur when these functions are handled by a general-purpose CPU. While some designs use ASICs to handle network functions, many system designers have moved toward using programmable *network processors* due to their increased flexibility and lower design cost.

In this paper, we describe a code generation technique designed for the *Agere Payload Plus* network processor. This processor utilizes a multi-block pipeline containing a Fast Pattern Processor (FPP) for classification, a Routing Switch Processor (RSP) for traffic management and a third block, the Agere Systems Interface (ASI), which provides additional functionality for performance. This paper focuses on code generation for the clustered VLIW compute engines on the RSP. Currently, due to the real-time nature of the applications run on the APP, the programmer must lay out and partition the application-specific data by hand to get good performance.

The major contribution of this paper is to remove the need for hand partitioning for the RSP compute engines. We propose both a greedy code-generation approach that achieves harmonic mean performance equal to code that has been hand partitioned by an application programmer and a genetic algorithm that achieves a harmonic mean speedup of 1.08 over the same hand-partitioned code. Achieving harmonic mean performance that is equal to or better than hand partitioning removes the need to hand code for performance. This allows the programmer to spend more time on algorithm development.

## Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*compilers, optimization*

## General Terms

Languages, Algorithms

## Keywords

Partitioning, scheduling, network processors

## 1. INTRODUCTION

Within the genre of special purpose network processors it is common for programmers to be directly responsible for specifying the data layout in order to achieve the tight schedules required to meet real-time constraints. In this paper we present algorithms that allow the compiler to perform such data layout without compromising code quality. We also apply those techniques to the Agere Payload Plus processor to experimentally evaluate our code generation methods.

Increasing network bandwidth requirements brought on by the dramatic growth of the Internet have forced system designers to move upper-layer network functions into network devices. With higher bandwidth standards, such as OC-192, general-purpose processors are unable to handle the desired functionality at wire speed. In order to maintain the desired bandwidth modern network devices use special-purpose hardware to provide the desired functionality.

Recent trends have moved away from using ASICs for these functions to increase the degree of flexibility by allowing the chips to be programmed and to reduce the cost of producing the chips. Vendors such as Intel, Agere and Motorola have produced network processors that are embedded in network devices to allow applications such as Service Level Agreements (SLAs) to be implemented at wire speed. Each of these processors uses a different architecture, trading off flexibility and ability to perform certain complex functions at wire speed.

Designs such as the Intel IXP architecture include many RISC processing engines and are programmed using C, providing a high degree of flexibility. The Agere Payload Plus (APP) design utilizes more specialized compute engines, reducing flexibility but using less power and providing the ability to do more complex traffic management functions at wire speed. Some previous work on compilation for network processors has focused on code generation for the Intel IXP architecture [15, 6]. In this paper, we look at code generation for the APP compute engines.

Since the APP is a special-purpose processor, it requires more programmer control than what is normally basic functionality when programming in C for a general-purpose processor. Specifically, when performing traffic management, traffic shaping or stream editing functions the language used to program the compute engines requires the programmer to lay out application-specific data in memory by hand. Some of the APP compute engines utilize a clustered VLIW design, where the functional units and registers are clustered in groups having limited connectivity between the groups, to allow parallelism with low cost and low power. Because of the clustering, how the data is laid out in the register file greatly affects the schedule length for the application. Since an application must perform its tasks at wire speed, making even a 1-cycle "mistake" can mean the program will not work within the processor's real-time constraints.

In this paper, we extend the language used for programming the compute engines of the APP to free the programmer from the need to perform data layouts. To support the language extensions, we present two algorithms for automatically generating the data layouts: a two-pass greedy algorithm and a genetic algorithm [14]. These algorithms make it possible for the programmer to achieve good performance without hand partitioning the data.

The remainder of this paper is organized as follows. Section 2 gives details on the APP architecture and the programming language used in writing applications for the compute engines. Section 3, presents recent work related to code generation for the clustered architectures. Section 4 describes a two-pass greedy code generation method and Section 5 details our approach using genetic algorithms. Section 6 presents our experimental evaluation. Finally, Section 7 gives our conclusions and discusses future work.

## 2. THE RSP AND C-NP

The Agere Payload Plus (APP) network processor consists of a multi-block pipeline containing a Fast Pattern Processor (FPP) for classification, a Routing Switch Processor (RSP) for traffic management and the Agere System Interface (ASI) for collecting statistics for traffic management. Figure 1 shows the organization of the APP.

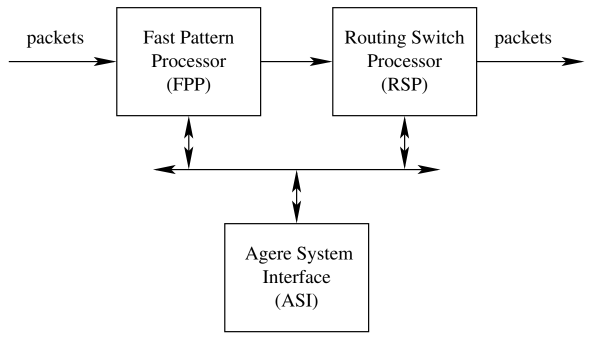
packets
Fast Pattern Processor (FPP)
Routing Switch Processor (RSP)
packets
Agere System Interface (ASI)

**Figure 1: Agere Payload Plus Architecture**

The RSP contains three compute engines, one each for stream editing, traffic management and traffic shaping. The compute engines are implemented using 4-wide clustered VLIW processors, that execute 2-address instructions. The clusters, henceforth called *slices*, each have a one-byte wide ALU and are named A, B, C and D. The RSP has 32 four-byte registers that are each cut into four one-byte slices. Each register slice may be accessed only by the ALU that has direct access to that slice. Thus, the first ALU may only access the first byte in each register and second ALU may only access the second byte in each register, etc. Registers are addressed by their register number and slice. For example, the third slice of register 14 is addressed as `R[14](c)`. Note that data longer than 8 bits cross slice boundaries. Operations of length 8- or 16-bits have no delay cycles. Figure 2 illustrates a single compute engine.

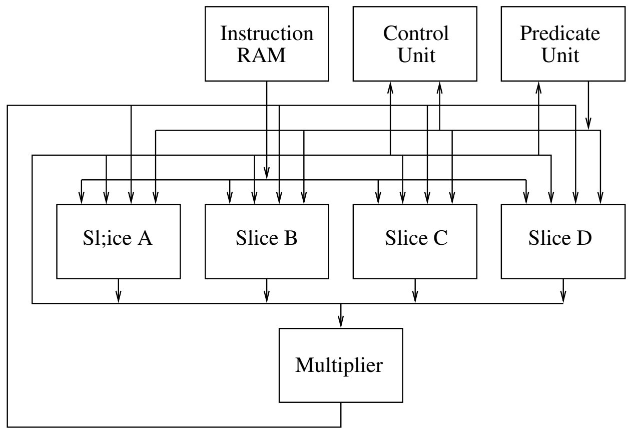

**Figure 2: RSP Compute Engine**

In order to facilitate two-address arithmetic and access to data in other slices, the RSP has two special registers: the Q and Y registers. The Q and Y registers can be used as temporary space when an operation uses more than two register addresses. This is because the Q and Y registers do not count against the two-address limit in the instruction word. In order to access data in another slice, the off-slice data must be copied into the Y register. The entire Y register may be *read* by any slice. It is the only register whose entire contents may be read by all slices. Access to the Q register is restricted by slice. Besides providing temporary space, as indicated previously, the Q register acts as a shift register when performing shift operations. The 128-byte register file plus the Q and Y registers constitute the *only data memory in the RSP compute engines.*

The compute engines of the RSP are programmed using C-NP. C-NP is a version of C specifically targeted for the Agere network processors. C-NP is restricted to assignment statements, if-statements and switch statements and is augmented to allow data placement of variables and direct byte addressing of the 128-byte register file. C-NP does not have facilities to access the Q or Y register. The programmer is responsible for data layout of application parameters. Since each slice only has access to a portion of the register file, effective data layout is essential to performance. If data is not in the portion of the register file directly accessible to the slice, it must be copied into the Y register before a computation can be performed, as mention previously.

In order to ease programming, we have introduced new syntax into C-NP that specifies a block of bytes within the register file where a particular variable may reside, allowing the compiler to determine which slice should have direct access to the variable. We call these variables *block free* since they are free to reside anywhere within a specified block.

As an example, consider the following code having a two-byte unsigned variable (x) beginning at the first byte of a 64-byte programmer-defined parameter area (param_block). The parameter area begins on byte 64 in the RSP register file.

```
block param_block[64:127];  // parameter area
unsigned x param_block[0:1];
```

With our additions to C-NP, the variable x can be declared as a block-free variable within the region of the register file defined by param_block as follows.

```
unsigned(2) x @param_block;
```

The compiler may assign x to any free 2-byte consecutive region between bytes 64 and 127 (registers 16 through 31), inclusive, in the compute engine register file. In the next section, we show how the compiler's choice of slice assignment affects the quality of code generated for the RSP.

## 2.1 A Partitioning Example

To see the effects of slice assignment on the length of a schedule, consider the following C-NP code:

```
block param_block[64:127];
unsigned(2) w,x,y,z @param_block;

w = w + x;
y = y + z;
```

If the compiler assigns w to bytes 64 and 65 (denoted as register R[16](ab)), x to bytes 66 and 67 (denoted R[16](cd)), y to bytes 68 and 69 (denoted R[17](ab)) and z to bytes 70 and 71 (denoted R[17](cd)), the compiler generates the code appearing in Figure 3. Note that the letters a, b, c and d in the register designations refer to the respective slices in Figure 2. The letter combinations ab and cd denote 16-bit data that cross slices a and b, and c and d, respectively.

```
Y(cd) = R[16](cd)
R[16](ab) += Y(cd)
Y(cd) = R[17](cd)
R[17](ab) += Y(cd)
```

**Figure 3: Example Intermediate Code**

This slice assignment yields two copy operations: one for x and one for z. Given a unit latency assumption for the operations, we can generate the schedule in Figure 4, where instructions in the same row are executed in parallel. The instructions on the left are executed in slices A and B and the instructions on the right are executed in slices C and D. Using this slice assignment, we get a schedule 3 cycles in length.

Since variables w and x are used together and variables y and z are used together, the compiler can put each pair in the same slice and reduce the schedule length. If the compiler puts w in bytes 64 and 65, x in bytes 68 and 69, y in bytes 66 and 67 and z in bytes 70 and 71, the compiler generates the 1-cycle schedule in Figure 5.

| | |
|---|---|
| | Y(cd) = R[16](cd) |
| R[16](ab) += Y(cd) | Y(cd) = R[17](cd) |
| R[17](ab) += Y(cd) | |

**Figure 4: Example Schedule 1**

| | |
|---|---|
| R[16](ab) += R[17](ab) | R[16](cd) += R[17](cd) |

**Figure 5: Example Schedule 2**

For the RSP, our compiler attempts to generate a schedule with as much parallelism as possible without introducing copy operations that cause an overall degradation in the schedule.

## 3. RELATED WORK

Considerable recent research has addressed the problem of generating code for general-purpose clustered architectures. Ellis [5] describes an early solution to this problem. His method, called BUG (bottom-up greedy), applies to a single scheduling context at a time (*e.g.,* a trace) and utilizes machine-dependent details and scheduling information within the partitioning algorithm.

Özer, et al., present an algorithm, called *unified assign and schedule* (UAS), for performing partitioning and scheduling in the same pass [11]. They state that UAS is an improvement over BUG since UAS can perform schedule-time resource checking while partitioning, allowing UAS to manage the partitioning with knowledge of the bus utilization for copies between clusters.

Desoli [4] describes a partitioning method that uses separate phases for assignment and scheduling. This method uses global pre-processing DAG analysis to reshape it for better performance and utilizes an iterative approach. Desolit reports good resuls for up to four clusters.

Nystrom and Eichenberger [10] present an algorithm that first performs partitioning with heuristics that consider modulo scheduling. Specifically, they prevent inserting copies that increase the recurrence constraint of modulo scheduling. If copies are inserted off the critical recurrences in recurrence-constrained loops, the initiation interval for these loops may not be increased if enough copy resources are available. Nystrom and Eichenberger report excellent results for their technique.

Hiser, et al. [7, 8], describe experiments with register partitioning in the context of whole programs and software pipelining. Their basic approach performs partitioning and scheduling in separate passes. Their method abstracts away machine-dependent details from partitioning with edge and node weights, a feature extremely important in the context of a retargetable compiler. Sule, *et al.* [13], use genetic algorithms to improve Hiser's approach. They show small improvements on a set of small kernels and applications.

Aletá, et al. [1], describe a technique that first partitions the data dependence graph and then schedules the instructions, respecting the cluster assignments made during partitioning. To enhance the partitioning phase, they make an initial partition, create a pseudo-schedule based on that partition and move nodes between partitions if improvements are obtained in the pseudo-schedule. The use of pseudo-schedules increases the amount of information available to the partitioning algorithm, resulting in better performance.

Leupers [9] uses simulated annealing to partition the data among registers and then uses list scheduling to evaluate the partition. His approach is similar to our approach using genetic algorithms. However, he does not need to deal with real-time constraints or fragmentation as we discuss later. Leupers reports good results on the TI C6201.

Two recent papers deal with code generation for the Intel IXP network processor. George and Blume [6] present an integer linear programming problem for modeling the constraints imposed on code generation for the IXP's dual-bank register file. Zhuang and Pande [15] present a heuristic technique of polynomial complexity to solve the same problem. Since the compute engines in the APP use two-address code and allow data to cross cluster boundaries, these results do not directly apply (although a modified formulation of the integer-linear programming solution proposed by George and Blume [6] could likely be used).

## 4. TWO-PASS PARTITION & SCHEDULE

In this section we describe our two-pass greedy algorithm for partitioning the data among the slices on the RSP. Our greedy approach is based upon the principle of unifying register assignment and scheduling for clustered VLIW processors, much like UAS [11]. We chose this approach because it allows the compiler to construct the schedule as partitioning occurs to give better information regarding open instruction slots. Compared with code generation for general-purpose clustered processors, code generation for the RSP compute engines includes three additional challenges. First, the code run on the RSP must meet real-time constraints (Audsley, *et al.* [3], discuss hard real-time scheduling for processes). If we cannot schedule a node within a fixed window, the APP will not be able to function at 100% of the line rate and packets will be dropped. In addition to using hard real-time deadlines in scheduling, we have chosen to use a two-pass algorithm that we call Two-Pass Partition and Schedule (TPPS). A one-pass algorithm such as UAS cannot take advantage of scheduling freedom obtained from knowledge of a complete schedule. Knowledge concerning empty instruction slots is only partially available during the first pass, preventing the scheduler from determining the actual cost of all copy operations. By applying a second pass to scheduling, we take advantage of holes in the schedule, where profitable, by moving data and or instructions between slices. Even a one-cycle improvement caused by modifying a register assignment may mean the difference between maintaining line rate or dropping packets. Second, our algorithm must consider register-file fragmentation because variables larger than 8-bits cross slice boundaries. Fragmentation leads to the possibility of partitioning failure due to a lack of enough consecutive space in the register file. Third, space for some variables is specified by a network protocol a priori, restricting the freedom of the partitioning algorithm.

The scheduling portion of TPPS is based upon standard list scheduling. The priority of each instruction is based upon three criteria. Instructions that have a small window available for legal schedule position due to real-time and semantic constraints have the highest priority. If no such instructions exist, then critical path length determines the instruction with the highest priority. Finally, if a tie still exists, the node with the largest number of successors has the highest priority. The priority for each node is updated dynamically as slice assignments and scheduling decisions are made for predecessor instructions.

The first pass of TPPS schedules the code with the goal of minimizing inter-slice communication. Since the ability to transfer data between slices is quite limited, we greedily try to choose a slice assignment yielding the lowest number of copies while providing parallelism where deemed profitable. The second pass of TPPS, looks at moving both block-free variables and instructions between slices to determine if the schedule can be shortened. This process takes advantage of the existence of a complete schedule where the holes that can be used to hide copy instructions are known. TPPS includes the following major steps, each of which will be discussed in detail below.

1. Compute initial information needed for scheduling and partitioning.
2. Select the best slices for any block-free variables in the instruction with the current highest priority.
3. Schedule the instruction from step 2 and go back to step 2 if any instructions remain.
4. Change the slice assignment of block-free variables if those changes shorten the schedule.
5. If fragmentation occurs, try a reference-priority based partitioning.

Figure 6 presents a description of TPPS. In the rest of this section, we elaborate on the steps of the algorithm.

### 4.1 Initialization

TPPS initially uses slice usage counts as a metric for copy costs. To compute the slice usage counts, each low-level instruction is examined for the occurrence of a block-free variable reference. If such a reference occurs, the other operands are examined to see if they already have slice assignments. The usage count for the block-free variable is incremented for the slices where the other assigned variables in the instruction are stored. Some instructions may only contain references to variables whose space has not been determined. These instructions provide no contribution to the usage costs. Note that at this point in compilation no instruction will contain references to variables in multiple slices. Our compiler performs a prepass that inserts all necessary copy operations into the Y register for all variables with a specific register file assignment before scheduling occurs. In addition, all intermediate instructions are converted to 2-address format using the Q and Y registers.

After usage counts have been computed, the algorithm computes two metrics for each low-level machine instruction: *as-soon-as-possible* (ASAP) and *as-late-as-possible* (ALAP). ASAP is the earliest time that a node may be legally scheduled and ALAP is the latest time that a node may be scheduled to meet real-time constraints. These times are continually updated as scheduling proceeds. The algorithm computes ALAP because there are real-time limits on how long a program can take. ALAP indicates the last possible cycle an instruction can execute and still have the schedule

```
procedure TPPS()
  compute slice usage counts for each block-free variable
  compute ASAP and ALAP for each node in the DDG
  add each DDG node to priority queue Q
  while Q not empty do
    n = Q.removeFirst()
    selectBestSlice(n)
    scheduleNode(n)
    add any newly ready nodes to Q
  endwhile
  adjustSliceAssignments()
  if fragmentation occurs
    try reference-count based partitioning
  pick the partition with the shortest schedule
end TPPS

procedure selectBestSlice(n)
  Let S be the set of slices
  for each block free variable b in n
    s == null
    while $S \neq \emptyset$ and s == null
      t =slice with lowest copy cost for b in S
      remove t from S
      if $\exists$ space in t for b and
          assigning b to t will not cause failure
        s = t
    end while
    if $s \neq$ null assign b to s
      else fail
  end for
end selectBestSlice

procedure adjustSliceAssignments()
  for each block-free variable b
    compute slice costs for each slice for b
    if $\exists$ a slice s with lower cost
          for b than current and space exists in s
      assign b to s
        add copies and reschedule instructions
      update usage counts
  end for
end adjustSliceAssignments
```

**Figure 6: Algorithm TPPS**

finish within a hard cycle limit. The difference between ASAP and ALAP is used in prioritizing the instructions: the smaller the difference between ASAP and ALAP, the higher the priority given to the instruction.

## 4.2 Slice Assignment

In the first pass, slice assignment is made based upon the usage counts computed for each slice. Before an instruction is scheduled, we make slice assignments for each block-free variable $b$ whose space has not been already assigned. Based upon usage counts, the algorithm chooses the slice $t$ that is likely to introduce the fewest copies. A copy is needed for any use in a slice other than that under consideration. If space for $b$ exists in $t$, then $t$ is chosen for $b$. If no space exists, the algorithm tries the next best slice. If more than one slice produces the fewest copies, the sum of the earliest available instruction slot in the slice and the schedule density of the slice is used to break the tie. The *schedule density* of a slice is the number of instructions currently scheduled in the slice plus those instructions that will be scheduled in the slice later because of the slice assignments of the operands. The density gives a lower bound on the number of cycles required by operations in a slice. The sum of earliest instruction slot and schedule density forces slice assignment to balance data between slices when one slice contains too many operations.

As mentioned previously, when choosing space for $b$ in $t$, the algorithm must consider whether assigning $b$ to $t$ might make it impossible to assign space for some other variable in any slice. Since each slice is only 8-bits wide, variables requiring more than 8-bits are laid out across multiple consecutive slices (this allows 16-bit operations to be completed in a single cycle). Having a variable cross slice boundaries may cause fragmentation to occur, possibly making it impossible to assign a larger variable to consecutive register locations. So, when making a slice assignment, we only choose a particular slice if fragments large enough for each remaining variable still exist. It may be impossible to choose a slice that does not cause a fragmentation problem. In this case, TPPS will fail. We discuss how failure is handled later.

Once $b$ has been assigned to some slice, the usage counts for any unassigned block-free variables may have changed. So, each instruction in which a newly assigned block-free variable occurs is examined to update the usage count of any remaining block-free variables in that instruction. Note that after slice assignment an instruction that previously had no preferred slice may now have one due to the assignment of a block-free variable.

## 4.3 Instruction Scheduling

After a slice is chosen for each block-free variable in an instruction, the instruction and any potential copies are scheduled. TPPS chooses the slice for the instruction based upon the sum of the earliest point the instruction can be scheduled given copy insertion and schedule density. Schedule density is used to help balance the operations across the slices. Ties are broken by schedule density within the slices alone. Slice choice for the instruction closely mirrors slice choice for block-free variables to allow the slice choice metric to have a predictable view of what will happen when the instructions containing the block-free variables are actually scheduled.

After scheduling an instruction, some successor instructions may need to be executed within a fixed number of cycles of the current instruction to ensure correct semantics. This can occur when instructions use transient resources set by the current instructions. To handle this situation, each DDG edge contains a minimum and maximum time. Normally, the maximum time is $\infty$. However, in cases where the instruction uses a transient resource, the maximum time indicates how long that resource is available. In cases where a successor of an instruction has a maximum time, we choose an instruction slot such that the successor with the max time has an available instruction slot within the fixed limit. This instruction is then given the highest priority in the next iteration of scheduling [2].

For the RSP, the compiler must split arithmetic instructions longer than 16-bits into multiple consecutive 16-bit or 8-bit operations. For these split instructions, a dependence chain is created with edges to successors in the chain having a max time of 1 cycle. This forces each operation to execute within one cycle of its predecessor. Lookahead scheduling [2] ensures that all operations in the chain can be scheduled in successive slots before a choice is made. This reduces the occurrence of scheduling failure when real-time constraints are enforced.

## 4.4 Slice Adjustments

After the first pass of the algorithm, a second pass attempts to determine if moving the slice assignment of any block-free variable will shorten the schedule. Because the compute engines in the RSP utilize two-address arithmetic, it is possible for the slice assignment computed in the first pass to require a copy operation into the Y or Q register even if all of the variables lie in the same slice. As mentioned previously, a reference to the Y or Q register does not count as one of the two addresses. So, if an instruction requires three addresses, one operand must be copied into the Y or Q register first. Since the first pass does not contain a complete schedule at the point where a slice assignment is made, we choose to wait until the second pass to handle this situation. Once a full schedule has been computed, we can determine if an empty slot exists for any necessary copy operations due to the two-address limit or if movement of a block-free variable to a new slice yields a better schedule.

Given a block-free variable, each instruction containing a reference to it is examined to compute the cost of moving the variable to every other slice and to compute the cost of the current slice due to the two-address limit. For each copy that may be required, if an open slot in a particular slice exists that does not require a delay of any successor operation, there is no cost in moving the block-free variable to the new slice for the current instruction. An open slot must be found between the latest predecessor of the current instruction and the earliest successor. If inserting a copy increases the number of live values that must be held in the same slices of the Y register beyond one, a penalty is given to this slice assignment equal to a save and restore operation for the Y register. Note also that if an instruction only contains references to the block-free variable under consideration and references to constants, the entire instruction can be moved without introducing copies. This is taken into account in determining the cost of slice movement.

Once the cost of slice movement is determined for every slice and every instruction containing a particular block-free variable, the algorithm chooses the slice with the lowest cost. If no space can be found in this slice, the next lowest slice is examined. This process continues until the original slice is chosen. After slice assignment is finalized, the algorithm inserts the required copy operations into the Y and Q registers and reschedules the instruction if necessary.

## 4.5 Reference-count-based Partitioning

If at any point in TPPS the best choice for a block-free variable is not used because of fragmentation, we try a second partitioning strategy on the program. In this strategy, we try to assign all block free variables to their respective best partitions using reference counts as a prioritizing function. The strategy assigns block-free variables to slices in the order of decreasing reference counts. The objective is to try to get the most used variables in their best slices. The schedule length of this partition is compared against the first TPPS partition and the better one is chosen.

As discussed previously, TPPS may fail to produce a legal slice assignment. This may occur when the region of the register file declared for a set of block-free variables is barely large enough (or exactly large enough) to hold all of the block-free variables, causing fragmentation to occur. When slice assignment fails, TPPS packs the block-free variables in the region consecutively irrespective of cost. This gives a legal, although likely poor, slice assignment. We chose to improve upon this approach using genetic algorithms as discussed in the next section.

## 5. GENETIC ALGORITHMS

Because of the heuristic nature of TPPS, it may fail to find the best possible assignment. It is often the case that the solution space for slice assignment is too large to consider an enumeration of all possible solutions. In these cases, stochastic methods such as simulated annealing, tabu search and genetic algorithms often yield excellent results [12]. These algorithms utilize randomness to escape local optima and increase the likelihood of finding a global optimum.

Genetic algorithms (GAs) use neighborhood search techniques. Like most other neighborhood search schemes, a GA scans through the search space looking for better solutions; however, it considers a set of solutions and has the capability of exploring multiple regions simultaneously. In addition, the algorithm scans the region containing a better subset of solutions more vigorously.

A GA generates an initial set of solutions which form the base population, essentially representing a starting point for the search. Let $S = \{s_1, s_2, s_3, \dots, s_n\}$ represent the set of solutions, or *population* a GA initially generates. $S$ consists of points in the search trajectory that may be extremely diverse if needed. Each solution present in the population is called a chromosome and is generally represented by a bit string encoding the information pertaining to the position of the solution in the search space and its degree of optimality.

Once a population has been generated, the GA iteratively applies the basic operators *crossover*, *reproduction* and *mutation*, to the population in order to scan the search space looking for better solutions. These genetic operators determine the GA's ability to converge to a good solution within a limited time. The GA iteratively applies these operators for a specified number of generations (iterations). The generation count is chosen such that a sufficient portion of the solution space will be examined.

## 5.1 Crossover

Crossover helps the GA generate new solutions and, to a large extent, determines the next point examined in the search space during the scanning process. Consider the following crossover example. Let

$$s_1 = \{a_1, a_2, a_3, a_4, \dots, a_n\}, \text{ and}$$
$$s_2 = \{b_1, b_2, b_3, b_4, \dots, b_n\}$$

represent two chromosomes in the population. Crossover operates by selecting points within the chromosomes at which the crossover would occur and possibly switching the contents of the two chromosomes between the crossover points. If in the chromosomes $s_1$ and $s_2$ points two and three were selected for crossover, after applying the operator two new solutions representing two possibly new points in the search space would be generated, namely:

$$s_3 = \{a_1, b_2, b_3, a_4, \dots, a_n\}, \text{ and}$$
$$s_4 = \{b_1, a_2, a_3, b_4, \dots, b_n\}.$$

Crossover may operate in several other ways [12]. However, in essence crossover tries to generate new *candidate* solutions which can be evaluated to see if a better solution has been obtained.

## 5.2 Reproduction

Reproduction selects the chromosomes from the population for crossover. Reproduction can be implemented simply through random selection of elements in the current population. Additionally, when new chromosomes are added to the populations, others are thrown out. GAs assign each chromosome a "fitness" based upon a function of the characteristics of the chromosome. As a GA generates new solutions to the population via reproduction, crossover and mutation, it throws out solutions with a lower "fitness" .

## 5.3 Mutation

While crossover and reproduction build newer population sets for the GA to explore, they are designed in ways such that the algorithm finally converges to a small area within the search space. Mutation brings about the necessary diversity in a localized population allowing the GA to possibly escape from local optima. Mutation randomly changes parts of a new chromosome under a given probability. For example if mutation decides to randomly change the third position of a chromosome represented as $s_1 = \{a_1, a_2, a_3, a_4, \ldots, a_n\}$ the new chromosome would be $s_1 = \{a_1, a_2, \overline{a}_3, a_4, \ldots, a_n\}$.

The three operators, *crossover*, *reproduction* and *mutation* represent the basic tools that the GA uses to explore the search space for a solution. By iteratively applying a combination of all of these operators to the population, the GA simultaneously scans diverse regions in the search space while at the same time exploring good solutions locally in detail where needed. Another interesting thing to note in the case of GAs is that the generation count is one of the *driver* functions deciding the quality of the final solution. In simpler words, the longer a GA is allowed to operate, the better the quality of the solution obtained. As the generation count increases the GA scans through larger portions of the search space thus being able to locate better solutions if and when they exist.

## 5.4 Properties

In contrast to many common heuristics, GAs have several attractive properties which, in part, account for their increasing popularity.

- The most important property of a GA is its ability to scan *non-linear* search areas with ease. The design of a GA is such that it can generate good results irrespective of the shape of the search area involved. This means that as long as the basic operators of a GA are properly built it can traverse a search space of any shape without facing any problems.

- The *mutation* operator gives a GA the capability to escape from a region within the search space if and when needed. This means that the probability of a GA getting stuck within a particular locality in the search space is negligible.

- Unlike most other heuristics, a GA is a parallel search technique. By operating on a set of solutions instead of a single solution, a GA is able to cover diverse regions in the search space simultaneously. This is a property that almost no other heuristic has.

- In spite of its capability to reach over vast regions of the search space, a GAs ability to scan promising neighborhoods more vigorously when compared to less promising regions of the search space is not diminished. In simpler words, a GA does not lose its quality of basically being a neighborhood search technique.

These properties provide significant power and flexibility in building an effective search strategy for a problem. However, as with most powerful general-purpose search methods, adjustment to solve specific problems is often required to improve search results. One significant advantage of GAs is that they can be easily adjusted by investigating combinations of a few general parameters such as population size, bias of selective reproduction methods, crossover method, and mutation rate. When applying a GA to a specific problem as described here, one typically experimentally evaluates several combinations of such parameters to see which provide good results for the given problem.

The computation time for GAs can make them prohibitive. However, in our context the programs being compiled tend to be small (see Table 1). Additionally, given real time constraints and the fact that the code will be embedded in a device and used for a long period of time, longer compile times can be tolerated.

## 5.5 Applying GAs to the RSP

To improve upon TPPS, we have chosen to use a genetic algorithm designed by Whitley [14] to search large spaces of potential solutions. We utilize a two-point crossover with random mutation of 0, 1 or 2 slice assignments in the bit string. Our implementation uses probabilities of 30% for a double mutation and 99% for at least a single mutation. The mutation rate should be set high enough to allow the GA to escape local optima. We tried several values for the mutation rates and these rates produced satisfactory results.

To construct the initial population for the GA, we use the solution generated by the two-pass algorithm and then generate random slice assignments to fill out the population. Each slice assignment for a block-free variable is converted to its binary equivalent and the binary strings for each block-free variable are concatenated to construct the bit representation for the slice assignment. For example, if the two block-free variables `x` and `y` are initially assigned to begin in slices A, denoted `00`, and B, denoted `01`, respectively, the solution is encoded as the bit string `0001`.

The fitness function that we have chosen is the number of cycles obtained by the Agere C-NP compiler for a given slice assignment. For each bit string representing a slice assignment, C-NP is emitted with the appropriate variable assignments. The Agere compiler is run on the emitted code and the cycle count is returned as the fitness for a string. In cases where a valid register assignment cannot be reached for a particular slice assignment a fitness of $\infty$ is returned. After a specified number of generations of the algorithm, the partition with the lowest cycle count is emitted.

During evaluation we discovered that in cases where fragmentation is a problem, the randomly generated solutions of the GA often were not legal solutions. In many cases the GA spent its entire time searching a space of invalid solutions. To compensate for this, upon recognition of an invalid solution, we randomly generate a permutation of the block-free variables and greedily pack the variables into consecutive register locations using the permutation as the order of assignment. This procedure greatly improved the performance of the GA on some test cases.

| **Kernel** | **Lines** | **B-free** | **Kernel** | **Lines** | **B-free** | **Kernel** | **Lines** | **B-free** |
|---|---|---|---|---|---|---|---|---|
| 1 | 99 | 10 | 17 | 45 | 14 | 33 | 53 | 5 |
| 2 | 78 | 9 | 18 | 102 | 14 | 34 | 56 | 4 |
| 3 | 88 | 10 | 19 | 51 | 1 | 35 | 59 | 6 |
| 4 | 130 | 10 | 20 | 34 | 1 | 36 | 94 | 11 |
| 5 | 68 | 9 | 21 | 33 | 1 | 37 | 62 | 6 |
| 6 | 62 | 8 | 22 | 49 | 1 | 38 | 113 | 11 |
| 7 | 63 | 8 | 23 | 50 | 2 | 39 | 77 | 7 |
| 8 | 73 | 4 | 24 | 68 | 4 | 40 | 83 | 8 |
| 9 | 73 | 6 | 25 | 90 | 2 | 41 | 81 | 3 |
| 10 | 71 | 4 | 26 | 106 | 8 | 42 | 90 | 9 |
| 11 | 69 | 8 | 27 | 84 | 7 | 43 | 120 | 10 |
| 12 | 75 | 8 | 28 | 88 | 6 | 44 | 97 | 6 |
| 13 | 55 | 8 | 29 | 95 | 8 | 45 | 110 | 13 |
| 14 | 66 | 10 | 30 | 77 | 6 | 46 | 40 | 5 |
| 15 | 68 | 8 | 31 | 66 | 5 | 47 | 78 | 7 |
| 16 | 74 | 8 | 32 | 82 | 6 | | | |

**Table 1: Kernel Characteristics**

For the experiment we used a population of size 100 and iterated for 1000 generations. We tried several sets of parameters for the GA and these parameters were chosen because they produced reasonable compile times and good results. If the number of possible solutions was less than 1024, we reverted to a brute-force search rather than using the GA. This search involves trying all possible partitions. However, it is still possible for the partitions to fail because of fragmentation. Rather than try all possible permutations of slice assignment, as with the GA we generate a random permutation and perform a greedy packing according to that permutation. Given that there are 4 possible solutions for each variable, brute-force search is used when there are fewer than 5 block-free variables in the application.

## 6. EXPERIMENTAL RESULTS

We have implemented the above algorithms in a source-to-source C-NP compiler. Our compiler reads in C-NP source, converts it to low-level intermediate, performs partitioning using the two-pass greedy approach and optionally using genetic algorithms. Once the partition for each variable has been determined, the compiler emits C-NP source containing the register assignments for all block-free variables. We then feed the automatically partitioned source through the Agere C-NP compiler version 3.0.285 to get the final generated scheduled code.

We have tested our implementation against a representative set of 47 C-NP kernels provided by Agere. Note that because of the real-time constraints of the domain only kernels execute on the RSP compute engines. Thus, a set of kernels represents an application. The set of kernels used in this experiment performs various traffic management, shaping, policing and stream editing functions. Due to the proprietary nature of the kernels, Table 1 reveals only the number of source lines (**Lines** columns) and the number of block-free variables (**B-free** columns) in each kernel. Note that in Table 1 the data is presented in three three-column sets.

To test our algorithm, we remove the hand-partitioned portion of the data stored in the user defined parameter area and re-declare the variables using our syntax introduced for block-free variables. We then run the modified kernels through our source-to-source C-NP compiler to generate an automatic partitioning. Finally, we compare the schedule lengths of hand-coded and compiler-generated versions of the kernel reported by the Agere compiler.

Table 2 presents the performance results of our two-pass greedy algorithm and genetic algorithm versus the original hand-partitioned kernels provided by Agere. The results are organized by high-level kernel function. While the number of copies introduced into the code is often of interest for general-purpose clustered VLIW processors to assess the effect of clustering, it is of less importance here. We are not interested in whether we can limit the effects of clustering, but rather in whether we can obviate the need for hand partitioning.

TPPS performs as well as the hand-coded version on 16 of the 47 kernels, better on 15 kernels and worse on 16 kernels. The harmonic mean speedup over all the kernels for TPPS is 1.00. For the kernels on which TPPS performs better, TPPS often found partitions that did not seem intuitive. One example is kernel 1, which contains two completely independent (both data and control) sections of code of approximately equal size. The hand-coded solution split the 16-bit data evenly between slices A and B, and slices C and D to induce the most parallelism. However, since the kernel stores its predefined 32-bit variables across all four slices beginning in slice A, TPPS puts almost all variables in slices C and D and produces a better schedule. Placing the 16-bit variables in slices C and D aligns them with the portion of the 32-bit variables that is used when the 16-bit variable is added to the 32-bit variable. The improved schedule resulted from fewer copies due to the mismatched alignment of the 16-bit and 32-bit variables in the hand-coded version.

Thirteen of the sixteen kernels (from the stream editing, traffic management and traffic shaping kernel sets) on which TPPS performs worse contain completely full parameter areas. The best partition produced by TPPS causes fragmentation, resulting in secondary choices for slice assignment. In three of these thirteen cases, TPPS could not find a legal partition and resorted to packing the variables into the space irrespective of preferred slices, giving poor results. The final three of the sixteen poor cases (kernels 4, 15 and 17) did not achieve hand-partitioned performance due to the source-to-source nature of our compiler. Given that we could only deduce the algorithms used by the Agere compiler from its output, we were limited in our ability to understand what

| **Kernel** | **TPPS Speedup** | **GA Speedup** |
|---|---|---|
| — *Traffic Policing Kernels* — | | |
| 1 | 1.04 | 1.04 |
| 2 | 1.06 | 1.06 |
| 3 | 1.05 | 1.05 |
| 4 | 0.95 | 1.05 |
| 5 | 1.06 | 1.06 |
| 6 | 1.00 | 1.00 |
| 7 | 1.00 | 1.12 |
| 8 | 1.25 | 1.25 |
| 9 | 1.05 | 1.25 |
| 10 | 1.00 | 1.00 |
| 11 | 1.00 | 1.05 |
| 12 | 1.00 | 1.00 |
| 13 | 1.00 | 1.00 |
| 14 | 1.00 | 1.00 |
| 15 | 1.00 | 1.00 |
| 16 | 0.89 | 1.14 |
| 17 | 0.97 | 1.10 |
| 18 | 1.08 | 1.08 |
| — *Stream Editing Kernels* — | | |
| 19 | 1.00 | 1.00 |
| 20 | 1.00 | 1.00 |
| 21 | 1.00 | 1.00 |
| 22 | 1.00 | 1.00 |
| 23 | 0.82 | 1.00 |
| 24 | 0.89 | 1.14 |
| 25 | 1.00 | 1.00 |
| — *Traffic Management Kernels* — | | |
| 26 | 0.87 | 1.05 |
| 27 | 0.94 | 1.21 |
| 28 | 0.90 | 1.20 |
| 29 | 1.00 | 1.05 |
| 30 | 1.10 | 1.22 |
| 31 | 0.86 | 1.00 |
| 32 | 0.93 | 1.00 |
| 33 | 1.33 | 1.33 |
| 34 | 1.20 | 1.20 |
| 35 | 1.22 | 1.22 |
| 36 | 0.83 | 1.25 |
| 37 | 1.18 | 1.18 |
| — *Traffic Shaping Kernels* — | | |
| 38 | 0.88 | 0.96 |
| 39 | 1.00 | 1.00 |
| 40 | 0.93 | 1.00 |
| 41 | 0.93 | 1.08 |
| 42 | 1.00 | 1.00 |
| 43 | 0.92 | 1.00 |
| 44 | 1.05 | 1.11 |
| 45 | 0.84 | 1.11 |
| 46 | 1.33 | 1.33 |
| 47 | 1.19 | 1.27 |
| **HMean** | 1.00 | 1.08 |

**Table 2: Performance in Machine Cycles**

to expect. We believe that implementation of TPPS within the Agere compiler will fix these last three cases.

The reference-count-based partitioning strategy used by our compiler when fragmentation prevents the best slice assignment normally performed worse than TPPS. Although it was invoked on all of the cases where TPPS made secondary slice assignments due to fragmentation, the reference-count strategy only produced better results on two kernels.

The GA achieved better performance than the hand-coded version on 28 kernels and better than TPPS on 22 kernels. The GA performed the same as the hand-coded version on 18 kernels and the same as TPPS on 25 kernels. Finally, the GA performed worse than the hand-coded version on 1 kernel. Overall, the GA achieved a harmonic mean speedup of 1.08 over the hand-partitioned code and 1.07 over TPPS.

In many cases where the GA improved performance, a partition with non-obvious (even less than those produced by TPPS) slice assignments produced the best schedule. In many cases, it is unlikely that a programmer would create a similar partition because the slice assignments do not appear to be good upon inspection. For the one kernel where the GA performed worse than the hand-partitioned version, there were very few valid solutions due to fragmentation. Overall the performance of the GA is excellent and its ability to find good solutions warrant its inclusion in a compiler for the RSP.

On average, the compile-time for the GA was 1500 times that of TPPS. Since the GA invokes the Agere compiler to evaluate the partition, it has additional overhead. Both our C-NP compiler and the Agere compiler are written in Java. However, the compile time for the GA was still kept to 10 minutes per kernel on a 2.4GHz Pentium IV with 512 MBytes of memory. We believe this is a reasonable overhead given the target domain.

## 7. CONCLUSIONS

In this paper, we have presented an algorithm called TPPS to automatically partition data for code run on the compute engines of the Routing Switch Processor for the Agere Payload Plus network processor. TPPS gives harmonic mean performance equal to the hand-coded kernels and sometimes performs even better. Thus, using simple heurstics the compiler can obtain performance as good as hand-coded applications. We have also shown that genetic algorithms improve upon TPPS. If one can tolerate longer compiler times, a genetic algorithm produces partitions that give a harmonic mean speedup of 1.08 over the hand-coded versions.

In the future, we plan to investigate better methods for dealing with fragmentation. We will look into using an integer linear programming formulation to deal with the entire partitioning and scheduling problem. In addition, we plan to extend our work to produce a development system that automates the process of partitioning a set of kernels among the compute engines on the RSP. This system will assist users in overall system design by determining the most profitable set of kernels that can be executed within the real-time constraints of the RSP.

Given the importance of the Internet, network processor applications are critical to the efficient processing of communication. Our work is an important step in making the development of network applications for the Agere Payload Plus easier.

## 8. ACKNOWLEDGMENTS

The first author was supported by a grant from Agere Systems, Inc. and by NSF grant CCR-0209036. The authors would like to thank Paul Hintikka of Agere for his help with understanding the compute engines in the RSP and the Agere C-NP compiler. Roger Bringmann of Agere provided the funding for this project.

## 9. REFERENCES

[1] A. Aletá, J. Codina, J. Sánchez, A. González, and D. Kaeli. Exploiting pseudo-schedules to guide data dependence graph partitioning. In *Procedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, Charlottesville, VA, September 2002.

[2] V. Allan, S. J. Beaty, B. Su, and P. H. Sweany. Building a retargetable local instruction scheduler. *Software Practice and Experience*, 28(3):249–284, March 1998.

[3] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard real-time scheduling: The deadline monotonic approach. In *Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software*, Atlanta, GA, 1991.

[4] G. Desoli. Instruction assignment for clustered VLIW DSP compilers: A new approach. HP Labs Technical Report HPL-98-13, HP Labs, Jan. 1998.

[5] J. R. Ellis. *A Compiler for VLIW Architectures*. PhD thesis, Yale University, 1984.

[6] L. George and M. Blume. Taming the IXP network processor. In *Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 26–37, San Diego, CA, June 2003.

[7] J. Hiser, S. Carr, and P. Sweany. Global register partitioning. In *Proceedings of the 2000 International Conference on Parallel Architectures and Compiler Techniques*, pages 13–23, Philadelphia, PA, October 2000.

[8] J. Hiser, S. Carr, P. Sweany, and S. Beaty. Register partitioning for software pipelining with partitioned register banks. In *Proceedings of the 14th International Parallel and Distributed Processing Symposium*, pages 211–218, Cancun, Mexico, May 2000.

[9] R. Leupers. Instruction scheduling for clustered VLIW DSPs. In *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*, pages 291–300, Philadelphia, PA, Oct. 2000.

[10] E. Nystrom and A. Eichenberger. Effective cluster assignment for modulo scheduling. In *Proceedings of the 31st International Symposium on Microarchitecture (MICRO-31)*, pages 103–114, Dallas, TX, December 1998.

[11] E. Özer, S. Banerjia, and T. Conte. Unified assign and schedule: A new approach to scheduling for clustered register file microarchitectures. In *Proceedings of the 31st International Symposium on Microarchitecture (MICRO-31)*, pages 308–316, Dallas, TX, December 1998.

[12] V. Rayward-Smith, I. Osman, C. Reeves, and G. Smith. *Modern Heuristic Search Methods*. John Wiley and Sons, Ltd., 1996.

[13] D. Sule, P. Sweany, and S. Carr. Evaluating register partitioning with genetic algorithms. In *Proceedings of the Fourth International Conference on Massively Parallel Computing Systems*, Ischia, Italy, April 2002.

[14] D. Whitley. An overview of evolutionary algorithms. *Journal of Information and Software Technology*, 43:817–831, 2001.

[15] X. Zhuang and S. Pande. Resolving register bank conflicts for a network processor. In *Proceedings of the 12$^{th}$ International Conference on Parallel Architectures and Compilation Techniques*, pages 269–278, New Orleans, LA, September 2003.