Optimizing Code Parallelization through a Constraint Network Based Approach*

Ozcan Ozturk, Guilin Chen, Mahmut Kandemir
Pennsylvania State University
University Park, PA 16802, USA
{ozturk,guilchen,kandemir}@cse.psu.edu

ABSTRACT

Increasing employment of chip multiprocessors in embedded computing platforms requires a fresh look at conventional code parallelization schemes. In particular, any compiler-based parallelization scheme for chip multiprocessors should account for the fact that interprocessor communication is cheaper than off-chip memory accesses in these architectures. Based on this observation, this paper proposes a constraint network based approach to code parallelization for chip multiprocessors. Constraint networks have proven to be a useful mechanism for modeling and solving computationally intensive tasks in artificial intelligence. They operate by expressing a problem as a set of variables, variable domains and constraints and define a search procedure that tries to satisfy the constraints (an acceptable subset of them) by assigning values to variables from their specified domains. This paper demonstrates that it is possible to use a constraint network based formulation for the problem of code parallelization for chip multiprocessors. Our experimental evaluation shows that not only a constraint network based approach is feasible for our problem but also highly desirable since it outperforms all other parallelization schemes tested in our experiments.

Categories and Subject Descriptors

D.3.4 [**Software**]: Programming Languages—*compilers*

General Terms

Performance

Keywords

chip multiprocessing, constraint network, compiler

1. INTRODUCTION

As chip multiprocessing is quickly becoming a viable approach for implementing complex single chip designs, software issues regarding the programming of the chip multiprocessors and the application code optimizations are becoming increasingly important. One of the issues that need to be tackled by compilers in the context of chip multiprocessors is automatic code parallelization. While code parallelization – in its different forms – have been around for more than two decades now, the problem requires a fresh look when considered in the context of chip multiprocessors. The main reason for this is the fact that interprocessor communication is not very expensive in chip multiprocessors and, one can tolerate some increase in interprocessor communication activity if it leads to a reduction in off-chip memory accesses, which are very costly (compared to interprocessor communication) in these architectures.

A compiler-based code parallelization targeting these architectures therefore needs to focus on cutting the number of off-chip requests. Unfortunately, most of the code parallelization techniques proposed and studied for high-performance parallel machines do not extend directly to chip multiprocessors due to two major factors. First, most of these code parallelizers handle one loop nest at a time and thus fail to capture the data sharing patterns across different loop nests which are important for minimizing off-chip accesses, as will be discussed later. Second, in parallelizing a loop nest, most of these previous compiler algorithms focus on minimizing interprocessor communication rather than optimizing memory footprint.

The main goal of this paper is to explore a new code parallelization scheme for chip multiprocessors used in embedded computing. This compiler approach is based upon the observation that, in order to minimize the number of off-chip memory accesses, the data reuse across different (parallelized) loop nests should be exploited as much as possible. This can be achieved by ensuring that a given processor accesses the same set of elements in different nests as much as possible. Unfortunately, this practically makes a solution that handles one loop nest at a time unsuitable. Instead, what is needed is an approach that considers the parallelization constraints across all the loop nests in the program code being optimized. The proposed approach achieves this by employing an optimization strategy based on *constraint networks*. Constraint networks have proven to be a useful mechanism for modeling and solving computationally intensive tasks in artificial intelligence [9, 17]. They operate by expressing a problem as a set of variables, variable domains and constraints (an acceptable subset of them) and define a search procedure that tries to satisfy all the constraints by assigning values to variables from their specified domains. This work shows that it is possible to use a constraint network based formulation for the problem of code parallelization for chip multiprocessors.

To show that this is indeed a viable and effective alternative to currently existing approaches to code parallelization, we implemented our approach and applied it to a set of five applications that can benefit a lot from chip multiprocessing. Our experimental evaluation shows that not only a constraint network based approach is feasible for our problem but also highly desirable since it outperforms all other parallelization schemes tested in our experiments.

The rest of this paper is organized as follows. The next section gives an example illustrating why a constraint network based approach makes sense in the context of code parallelization for chip multiprocessors. Section 3 gives the technical details of our ap-

*This work is supported in part by NSF Career Award #0093082, and a grant from GSRC.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC 2006, July 24–28, 2006, San Francisco, California, USA.
Copyright 2006 ACM 1-59593-381-6/06/0007 ...$5.00.

```
for(i = 1; i ≤ N; i++)
  for(j = 1; j ≤ N; j++)
    A[i][j] = A[i][j] + B[j][i] * C[i][j];

for(i = 1; i ≤ N; i++)
  for(j = 1; j ≤ N; j++)
    D[i][j] = D[i][j] - B[i][j];

for(i = 1; i ≤ N; i++)
  for(j = 1; j ≤ N; j++)
    B[j][i] = B[j][i] * C[i][j];
```

Figure 1: Example program fragment.

proach. It first presents the problem formulation and then discusses the solution strategy we adopted. Section 4 presents an experimental evaluation of our approach and compares it quantitatively to three alternate code parallelization schemes. Section 5 revises the related work on code parallelization. Section 6 concludes the paper by summarizing our major results and briefly discusses the ongoing work.

2. MOTIVATING EXAMPLE

In this section, we discuss using an example code fragment why a constraint network based approach can be a desirable option over the known code parallelization techniques. For this purpose, we consider the code fragment shown in Figure 1. In this code fragment, three loop nests manipulate a set of arrays and these nests share arrays B and C, which are the focus of our attention. Let us assume, for ease of illustration, that we are allowed to parallelize a single loop from each loop nest of this program fragment and that we use four processors to parallelize each nest. Figure 2 depicts the sections of arrays B and C that are accessed by the processors. The sections accessed by the first processor are in a different color than the others. If we do not care about data sharing among different loop nests, one may choose to parallelize only the i loops from each nest.[1] However, it is not difficult to see that, under such a parallelization scheme, the first processor accesses different segments of arrays B and C in the first and second loop nests (this is true for the other processors as well). Consequently, as the execution moves from the first loop nest to the second one, this processor will make little use of the contents of the on-chip caches. Similarly, in moving from the second nest to the third one, again, the different sections of the arrays are accessed. While it is true that the first and third nests access the arrays in the same fashion (i.e., access the same array sections), the intervening nest (the second one) can easily destroy cache locality if the arrays at issue are large enough. The main problem with such a nest centric parallelization strategy is that it fails to capture the data sharings among the different loop nests. What we would prefer, in comparison, is a parallelization scheme such that a given processor accesses the same set of array segments during the execution of the different nests of the program. In our example in Figure 1, such a parallelization strategy would select one of the following two alternatives: (a) parallelize the i loops from the first and third nests and the j loop from the second nest, or (b) parallelize the j loops from the first and third nests and the i loop from the second nest. It needs to be noted that, under any of these two alternatives, a processor accesses the same segments of arrays B and C in all three loop nests shown. In order to derive such parallelizations, we need an approach that captures the data sharing across the different loop nests and considers all loop nests at the same time (not in isolation). In the rest of this paper, we present such an approach built upon the constraint network paradigm.

[1]Note that none of the loops in this code fragment carries any type of data dependence that prevents the parallel execution of its iterations. Thus, the compiler has a complete flexibility in selecting the loops to run parallel.

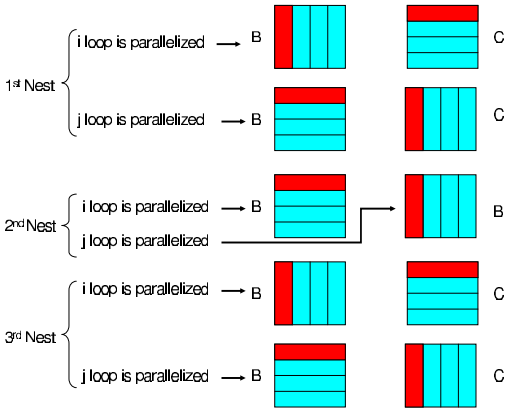

Figure 2: Array partitionings under different parallelization strategies for the example code fragment in Figure 1.

3. CONSTRAINT NETWORK BASED PARALLELIZATION

3.1 Preprocessing

Before building our constraint network that captures all the constraints necessary for determining the loops to parallelize in a given program, we need to identify what type of data (array) access pattern is obtained after each loop parallelization. What we mean by "data access pattern" in this discussion is the style using which the array is partitioned across the parallel processors. As an example, let us consider the access pattern for array B in the first loop nest of the example in the previous section. When the i loop is parallelized, each processor accesses a set of consecutive columns of this array and we can express this access pattern using $[*, block(4)]$, which means that the second dimension of the array is distributed over 4 processors, whereas the first dimension is not distributed. Similarly, the access pattern for the same array when the j loop is parallelized (instead of i) can be captured as $[block(4), *]$. If, on the other hand, for a three-dimensional array in a different example, the first, second, and third dimensions are distributed over 3, 2, and 6 processors, respectively, the resulting data access pattern can be specified as $[block(3), block(2), block(6)]$. Capturing access patterns on different arrays is important because if two different loops, say i and j, of two different nests have the same access patterns on the same set of arrays, these two loops are good candidates for parallelization. The job of the constraint network based approach described in the next subsection is to search for such loops – throughout the program code – that, when parallelized, they result in the same access pattern. This in turn means that a processor accesses the same array segments when these two loops are parallelized. Consequently, we can expect a good data locality (on-chip cache behavior) from such an access pattern.

3.2 Problem Formulation

A *constraint network (CN)* can be described as a triple $CN = < \mathcal{P}, \mathcal{M}, \mathcal{S} >$, where $\mathcal{P}$ is a finite set of variables, $\mathcal{M}$ is a list of possible values for each variable (i.e., its domain), and $\mathcal{S}$ is a set of constraints on $\mathcal{P}$ [8]. In this paper, $\mathcal{P}$ holds the loop nests in the application code being parallelized, i.e., each element of $\mathcal{P}$ represents a loop nest in the application code. $\mathcal{M}$, on the other hand, holds the possible (legal) parallelization patterns for the loop nests. For a given loop nest $n_i \in \mathcal{P}$ with L loops, each element of the corresponding entry in $\mathcal{M}$, say m_i, is an L-bit sequence where the j^{th} bit indicates whether the j^{th} loop of nest n_i is parallelized ($L(j)$=1) or not ($L(j)$=0). Note that while a nest n_i with L loops can potentially have 2^L possible bit sequences (different combinations of 0 and 1) in m_i, many of these sequences would not be acceptable in practice. For example, due to data and control de-

pendences, sometimes it is not possible to parallelize a certain loop in a given nest. In this case, we can drop all the sequences with the associated bit set to 1. Also, in some cases, we may not want to exploit a very fine grain parallelism in the loop nest, that is, we may want to limit the number of loops that can run in parallel. This can be achieved by dropping the sequences with certain number of 1s. The last component of our constraint network, $\mathcal{S}$, captures the constraints that need to be satisfied for maximizing data reuse among different loop nests. As stated above, maximizing data reuse among loop nests is important for reducing the number of off-chip data references, which in turn helps reduce both power consumption and execution cycles. An entry of $\mathcal{S}$, say s_k, is of the form $bs_{k1}, bs_{k2}, \cdots, bs_{kq}$, where each bs_{kp} - where $1 \leq p \leq q$ - is a bit sequence for a nest in the application. What an entry such as s_k indicates to the constraint network that, for exploiting inter-nest data reuse, the corresponding nests should have the bit sequences $bs_{k1}, bs_{k2}, \cdots, bs_{kq}$. We want to emphasize that an $\mathcal{S}$ set may have multiple entries and we need to satisfy only one of them.

Let us now consider an example that illustrates how we can build a constraint network for a specific example. We consider again the program fragment given in Figure 1. The corresponding constraint network can be written as follows under the assumption that we are allowed to parallelize only a single loop from each nest:

$$\begin{aligned} CN &= <\mathcal{P}, \mathcal{M}, \mathcal{S}>, \quad \text{where} \\ \mathcal{P} &= \{n_1, n_2, n_3\}; \\ \mathcal{M} &= \{\{(1,0),(0,1)\}, \{(1,0),(0,1)\}, \{(1,0),(0,1)\}\}; \\ \mathcal{S} &= \{\{(1,0),(0,1),(1,0)\}, \{(0,1),(1,0),(0,1)\}\} \end{aligned}$$

We now discuss briefly the components of this constraint network formulation. First, the $\mathcal{P}$ set says that there are three loop nests (n_1, n_2, and n_3) in the program fragment under consideration. The domain set, $\mathcal{M}$, on the other hand, indicates the potential values (parallelization strategies) that each loop nest can assume. For example, the first entry in $\mathcal{M}$, $m_1 = \{(1,0),(0,1)\}$, tells us that, as far as nest n_1 is concerned, there are two possible parallelization strategies. The first strategy, captured by bit sequence $(1,0)$, corresponds to the case where the first loop is parallelized and the second one is run sequentially (i.e., not parallelized). The second strategy, captured by $(0,1)$, represents an opposite approach where the second loop is parallelized and the first loop is to be executed serially. The reason why we have only two options (two alternate parallelization strategies) for the first loop nest is our assumption above which states that we can parallelize only a single loop from each nest. Relaxing this assumption (i.e., allowing the possibility of parallelizing both the loops) would modify the corresponding entry for n_1 in $\mathcal{M}$ to $\{(1,0),(0,1),(1,1)\}$. We use domain variable m_i to represent the possible (allowable) parallelization strategies for a nest n_i. In our running example, $m_1 = m_2 = m_3 = \{(1,0),(0,1)\}$, which means all three loop nests have the same flexibility as far as parallelization is concerned.

We now explain the constraint set $\mathcal{S}$. This set essentially captures the constraints for maximizing the data reuse across the different nests from a processor's perspective. Let us first focus on the entry $\{(1,0),(0,1),(1,0)\}$. What this entry specifies that in order to maximize the inter-nest data reuse, one option is to parallelize the first loop from the first nest, captured by bit sequence $(1,0)$; the second loop from the second nest, captured by $(0,1)$; and the first loop from the third nest, captured by $(1,0)$. The other entry in $\mathcal{S}$, $\{(0,1),(1,0),(0,1)\}$, can be interpreted in a similar way. The obvious question is how one can come up with these entries, i.e., how we can populate $\mathcal{S}$. The answer to this question lies in the explanation given in Section 3.1. When we detect the fact that parallelizing the first loop, second loop and first loop from the first nest, second nest and third nest, respectively, allows a processor to access the same array segments, we build the entry $\{(1,0),(0,1),(1,0)\}$ and put it in the set $\mathcal{S}$.

It needs to be emphasized at this point that our job is to select a parallelization strategy for each loop nest, i.e., to choose an entry from m_i for n_i, such that at least one of the entries (s_j) in $\mathcal{S}$ is completely satisfied. For example, if we select $(1,0)$ from m_1, $(0,1)$ from m_2, and $(1,0)$ from m_3, we satisfy the first entry of $\mathcal{S}$ – $\{(1,0),(0,1),(1,0)\}$. While, in this simple example, it is relatively straightforward to identify the required selections, it is not difficult to imagine that, for a large embedded application with tens of loop nests and arrays, this is not a trivial problem to solve. In the next section, we discuss an automated search strategy for this problem.

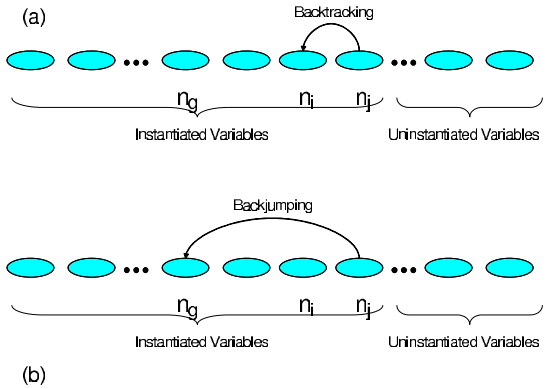

Figure 3: Comparison of backtracking (a) and backjumping (b).

3.3 Proposed Solution

We now discuss a backtracking based solution to the problem of determining a suitable parallelization strategy for a given program. Our solution operates on the constraint network based representation of the problem explained above. We later discuss several enhancements to our backtracking based approach that aim at minimizing the time to find a solution. Recall from our discussion in the previous section that our task is to assign, for each nest n_i, a value from the corresponding m_i domain such that at least one of the entries, say s_j, of $\mathcal{S}$ is completely satisfied. Here, we have $1 \leq i \leq V$ and $1 \leq j \leq W$, where V is the number of loop nests in the application program and W is the number of possible constraints, any of which can be satisfied.

In a given constraint network, a *partial instantiation* of a subset of variables is an assignment to each variable from its domain. A *consistent partial instantiation,* on the other hand, is a partial instantiation that satisfies all the constraints (of an s_j in our problem) that involve only the instantiated variables [8]. A *backtracking* based algorithm basically traverses the state space of partial instantiations in a depth-first manner. It starts with an assignment of a variable (e.g., randomly selected) and then increases the number of partial instantiations. When it is found that no solution can exist based on the current partial instantiation, it backtracks to the previous variable instantiated, and re-instantiates it with a different value from its domain. Therefore, a backtracking algorithm has both *forward* (where we select the next variable and instantiate it with a value) and *backward* phases (where we return to the previously instantiated variable and assign a new value to it). In the rest of the paper, this backtracking based scheme is referred to as our *base scheme*. In our code parallelization problem, the base scheme operates as follows. We first select a parallelization strategy from among the strategies specified in the m_1 set for the first nest (say n_1). All s_js that cannot accept this assignment are dropped from consideration immediately. If there exists at least an s_j that can accept this assignment, we continue by selecting a parallelization for n_2, the second nest. If there still exists at least an s_j that is consistent with the assignments made so far (i.e., a consistent partial instantiation), we go ahead with the third nest, and so on. At some point, if we determine that there is no s_j that is partially consistent with the assignments made so far, we backtrack by changing the parallelization selection made for the last nest considered and

check for consistency again, and so forth. At the end of these forward and backward movements (phases), we have either of two outcomes. We either satisfy an s_j completely (i.e., full instantiation) or we fail to satisfy any s_j. In the latter, we can relax the constraints in $\mathcal{S}$ or add some more entries (alternate s_js) to $\mathcal{S}$ and retry.

Let us apply the backtracking based solution to the example given in Figure 1. Based on the $\mathcal{P}$, $\mathcal{M}$, and $\mathcal{S}$ sets defined earlier for this code fragment, we first select $(1,0)$ from m_1 for the first nest, n_1. Since this assignment will be consistent with only s_1, s_2 will be dropped and we will proceed with the second loop nest. We continue by selecting $(1,0)$ from m_2 for n_2, which will only be consistent with s_2. Since s_2 was dropped in the previous step, there is no s_j that is partially consistent with this selection. At this point, we backtrack and select a different entry for n_2 from m_2, that is, we continue with $(0,1)$. This will be partially consistent with s_1 = $\{(1,0),(0,1),(1,0)\}$. So far, $(1,0)$ and $(0,1)$ have been selected for the first and the second loop nests, respectively, which is consistent with s_1. Next, we select $(1,0)$ from m_3 for the last loop nest, n_3. Again, this will be consistent with s_1 resulting in a full (consistent) instantiation, which satisfies s_1 completely. Hence, a suitable parallelization strategy is determined for the given code fragment by selecting $(1,0)$ from m_1, $(0,1)$ from m_2, and $(1,0)$ from m_3.

The important point to note is that the base scheme explained above makes random decisions at several points. The first random decision is to select the next variable (array) to instantiate during the forward phase. The second random decision occurs when selecting the value (parallelization strategy) with which the chosen variable is instantiated (again in the forward phase). In addition, in our base scheme, when we find out that the current instantiation cannot generate a solution, we always backtrack to the previously assigned variable, which may not necessarily be the best option. One can improve all these three aspects of the base scheme as follows. As for the first random decision, we replace it by an improved approach that instantiates, at each step, the variable that maximally constrains the rest of the search space. The rationale behind this is to be able to detect a dead-end as early as possible during the search. Similarly, when selecting the values to be assigned to the instantiated variables, instead of selecting a value randomly, we can select the value that maximizes the number of options available for future assignments. The rationale behind this is to increase the chances for finding a solution quickly (if one exists). Finally, we can expedite our search by *backjumping*, i.e., instead of backtracking to the previously instantiated value, we can backtrack further when it is beneficial to do so.

Backjumping in our approach can be best explained using an example scenario. Suppose that, in the previous step, we selected a parallelization strategy $x \in m_i$ for loop nest n_i, and in the current step, we selected a parallelization strategy $y \in m_j$ for loop nest n_j. If, at this point, we see that there cannot be any solution based on these assignments (i.e., we cannot satisfy any $s_k \in \mathcal{S}$), our base approach explained above backtracks to nest n_i and selects a new alternative for it (i.e., tries a new parallelization strategy, say $z \in m_i$, for it), assuming that we have already tried all alternatives for loop nest n_j. However, it must be noted that, if there is no constraint in the network in which both n_i and n_j appear together, that is, these loop nests do not share any array between them, assigning a new value (parallelization strategy) to n_i would not generate a solution, as n_i cannot be the culprit for reaching the dead-end. Instead, backjumping skips n_i and determines a loop nest (say n_g) among the loop nests that have already been instantiated that co-appears with n_j in a constraint, and assigns a new parallelization strategy to it (i.e., different from its current value). In this way, backjumping can prevent useless assignments that would normally be performed by backtracking and, as a result, expedite our search. Figure 3 gives an illustration that compares backtracking and backjumping. In the remainder of the this paper, our base approach supported by these three improvements is referred to as the *enhanced scheme.* The next section presents experimental data for both the base and enhanced schemes. Before going into our experimental analysis though, we need to make one point clear. If a solution exists to the problem under consideration, both the base and enhanced schemes will find it. However, if multiple solutions exist, they can potentially find different solutions.

Table 1: Benchmark codes.

Benchmark	Explanation	Domain Size	Data Size (KB)	Compilation Time (msec)	Execution Time (sec)
Med-Im04	Medical Image Reconstruction	311	825.55	812.52	204.27
Laplace	Laplace Equation Solver	887	1,173.56	867.30	69.31
Radar	Radar Imaging	497	905.28	690.74	192.44
Shape	Pattern Recognition and Shape Analysis	761	1,284.06	1,023.46	233.58
Track	Visual Tracking Control	423	744.80	785.62	231.00

4. EXPERIMENTAL EVALUATION

For each benchmark code in our experimental suite, we generated five different versions. The first version, called the original version, is the unmodified/unparallelized code executed on a single processor. The results with all other versions are given as speedups over this version. These other versions are parallelized codes. The second version we tested parallelizes each loop nest in isolation and is referred to as the nest based parallelization in this section. The third and fourth versions improve over the second one by carrying out some information from one loop nest to another. For example, when a loop nest is being parallelized, we record how the processors access the arrays referenced in the nest, and use this information in parallelizing the next loop nest. For example, if the first loop nest results in an access pattern represented by $(block(8), *)$ for an array, the parallelization strategy attempted in the second loop nest tries to achieve the same access pattern for the same array. These two versions, named global (first nest) and global (ordered), differ from each other in the order at which the loop nests are processed. Specifically, in the global (first nest) version, the loop nests are processed (parallelized) starting from the first one and ending with the last one, according to their textual order in the program. In comparison, in the global (ordered) version, we first determine an order of processing for the loop nests and then process them according to this order (note that this is just an order for processing the nests and we do not modify the textual positions of the nests with respect to each other). The order (of processing) used in our current implementation is based on the estimated cycle count for the nests (which is obtained through profiling). That is, we parallelize the loop nests one-by-one starting with the most expensive one and ending with the least expensive one. The last version evaluated in our experiments is the constraint network (CN) based approach proposed in this paper (our base scheme). As explained earlier, it first formulates the problem on a constraint network and then finds a solution based on a backtracking based search strategy.

We made our experiments using a SimpleScalar based infrastructure [2]. Specifically, we spawned a CPU simulation process for simulating the execution of each processor in our chip multiprocessors and a separate communication simulation process captured the data sharing and coherence activity among the processors. Each processor in the system has been modeled as a simple embedded core (300MHz) that can issue and execute two instructions at each clock cycle. Each processor has separate L1 instruction and data caches (each is 16KB, two-way set-associative with a line size of 32 bytes). The assumed latencies for the on-chip caches and the off-chip memory are 2 cycles and 90 cycles, respectively.

The benchmark codes used in our experiments are given in Table 1. The third column shows the total search space size (i.e., the sum of the domain sizes of all the loop nests in the application). The fourth column gives the total data size manipulated by each application (sum of all the data arrays). The last two columns give the compilation time and execution time of each benchmark under the original version. We implemented our constraint network using

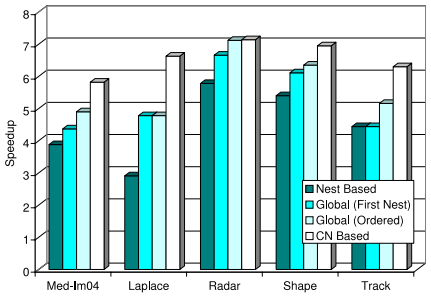

Figure 4: Speedups with different versions over the original version (8 processors).

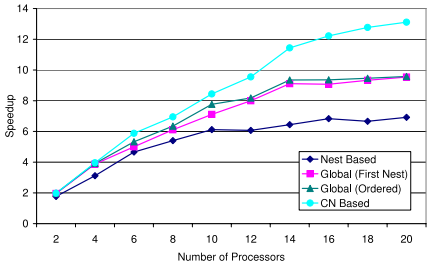

Figure 5: Influence of the number of processors (Shape).

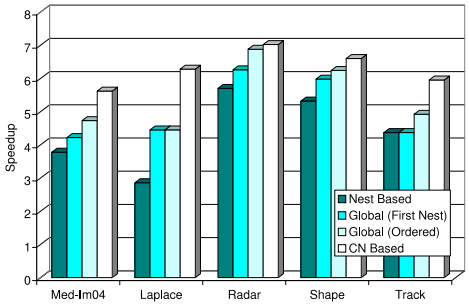

Figure 6: The speedup results with 64KB per processor data caches.

the C++ programming language. Excluding the libraries linked and comment lines, this implementation is about 2,050 C++ lines.

We first present in Figure 4 the speedups under four optimized versions of each benchmark when 8 processors are used for execution. We compute the speedup as $T/T'(p,v)$, where T is the execution time of the original version (see the last column of Table 1) and $T'(p,v)$ is the parallel execution time of version v under p processors. One of the important results from this bar-chart is that our constraint network based approach generates the best execution times for all the five benchmarks. The average speedups with the nest based, global (first nest), global (ordered) and our approach are 4.48, 5.72, 5.66 and 6.57, respectively. In addition to this main result, we also observe some other interesting trends from this graph. For example, the loop nest based parallelization strategy results in the worst performance among all the parallel versions we have, meaning that it is very important to capture the interactions between the different loop nests (this is a direct result of array sharing across the loop nests in our applications). In comparison, the versions global (first nest) and global (ordered) perform better than the nest based version. In fact, except for the benchmark Track, the global (first nest) version generates better results than the nest based parallelization. Another interesting result we see is that, except for the benchmark Laplace, global (ordered) perform better than global (first nest), indicating that it is also critical to select a good order to process the loop nests. This is because if we do not start processing with the most expensive nest, it is very likely that we will have a lot of constraints to satisfy when we come to process it. Our constraint network based approach does not have this problem because it determines a solution that satisfies the needs of all the loop nests (if such a solution actually exists). On the other hand, the reason why our approach generates better results than the global (ordered) scheme is that the latter does not optimize all the nests at the same time. Although it favors the optimization of the costly nests over the cheaper ones, in many cases, under the global (ordered) scheme, a number of cheaper nests go unoptimized for the sake of optimizing a single costly nest. Even worse, in some other cases, there are two (almost) equally expensive loop nests and the decisions made by the global (ordered) scheme in optimizing one of them can restrict the potential optimization opportunities for the other.

While the improvements in execution time presented above are certainly important and encouraging, for a fair comparison of the different versions, we need to look at the time they spend in finding a solution as well. Table 2 gives (in seconds) the solution times for the different methods. We see from these results that, as expected, our approach takes more time to find a solution than the remaining three optimized (parallel) versions. However, as we will discuss shortly, we can cut this solution time dramatically by adopting a more intelligent search and variable/value selection procedure.

We next look at the behavior of the different versions when the number of processors is changed. While we present results in Figure 5 only for the benchmark Shape, the observations we make extend to the remaining four benchmarks as well. We see that the

Table 2: Solution times taken by different versions. All times are in seconds.

Benchmark	Nest Based	Global (First Nest)	Global (Ordered)	CN Based
Med-Im04	6.89	9.18	9.33	66.84
Laplace	9.53	12.16	12.83	51.07
Radar	5.97	8.07	8.61	72.59
Shape	9.50	11.74	12.13	82.94
Track	8.54	10.93	11.35	64.30

scalability exhibited by the nest based, global (first nest) and global (ordered) versions are not very good (as we increase the number of processors). This is mainly because of the increased amount of interprocessor data sharing activity which becomes really a dominant factor beyond a certain number of processors. Our approach on the other hand effectively increases the total on-chip cache capacity by maximizing the data reuse across the different loop nests in these applications and this in turn enhances it scalability substantially.

To see what happens when we have more on-chip storage, in our next set of experiments, we increased the per processor L1 capacity to 64KB (recall that the default value used so far was 16KB per processor). The speedup results are presented in Figure 6 (for the 8 processor case). When we compare these results with those presented in Figure 4, we see that the speedups are reduced for all the parallel versions. The main reason for this behavior is that the original version takes a great advantage of the increased on-chip storage (from 16KB to 64KB for that version) and, as a consequence, the relative speedups achieved by the parallel versions are reduced. We also observe however that, even with a 64KB data cache per processor, the average speedups with the nest based, global (first nest), global (ordered) and our approach are 4.41, 5.07, 5.44 and 6.29, respectively. In other words, our approach still generates the best results among all the versions tested.

Recall that we discussed in Section 3.3 three possible enhancement to our base approach. The first enhancement is to do with the selection of the variable to instantiate next; the second one is related to the selection of the value to be assigned to the selected variable; and the last one is to employ backjumping instead of backtracking. We now quantify the benefits coming from these three enhancements. Our experiments with these enhancements showed that they do not have a significant impact on the solution found (though, in some cases, the solutions found by the base and enhanced versions of our approach differ slightly); therefore, the enhanced version generates almost the same execution time results as our base version used so far in our experimental evaluation. However, these three enhancements impact the solution times significantly. Figure 7 presents the reduction in solution times over the last column of Table 2. The solution time reduction is in the range of 61% – 70% when considering all five benchmarks. In addition, each bar in Figure 7 is divided into three parts, each corresponding to one of the three potential optimizations discussed in Section 3.3. While, according to these results, most of the benefits come from backjumping, all three enhancements are very useful in general and

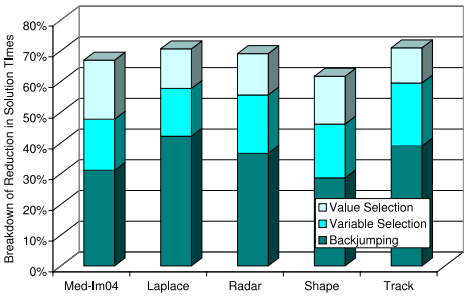

Figure 7: Reductions in solution times brought by the enhanced version over the base version.

contribute significantly to the overall reduction in solution times, without affecting the quality of the generated output code in any way.

5. RELATED WORK

In this section, we revise the related work on automatic code parallelization and compare it to the approach proposed in this paper. Bondalapati propose techniques for parallelizing nested loops that appear in the digital signal processing domain [4]. To parallelize such loops, they exploit the distributed memory available in the reconfigurable architecture by implementing a data context switching technique. Virtex and the Chameleon Systems have been used to implement the propsed methods. In [12], Kadayif et al exploit the use of different number of processor cores for each loop nest to obtain energy savings. In the proposed approach, idle processors are switched to a low-power mode to increase the energy savings. To reduce the performance penalty, a pre-activation strategy is also discussed. Goumas et al [10] propose a framework to automatically generate parallel code for tiled nested loops. They have implemented several loop transformations within the proposed approach using MPI, a message-passing parallel interface. A modulo scheduling algorithm to exploit loop-level parallelism for coarse-grained reconfigurable architectures has been proposed by Mei et al [14]. Hogstedt et al [11] investigate the parallel execution time of tiled loop nests. They use a prediction formula for the execution time of tiled loop nests to aid the compiler. Using this prediction, compiler is able to automatically determine the tiling parameters that minimizes the execution time. Navarro et al [15] target minimizing communication and load imbalance in parallel programs. They represent the locality using a locality graph and mixed integer nonlinear programming is used on this graph to minimize the communication cost and load imbalance. Parallelism in a nested loop is exploited in [5] by analyzing data dependences. A variable renaming technique is used to break anti and output dependences along with a technique to resolve recurrences in a nested loop. Arenaz et al [1] propose a gated single assignment (GSA) based approach to exploit coarse-grain parallelism in loop nests with complex computations. They analyze the use-def chains between the statements of a strongly-connected component, as well as between the statements of the different strongly-connected components. Yu and D'Hollander [18] present a 3D iteration space visualizer to analyze the parallelism in nested loops. An iteration space dependency graph is constructed to show the data dependencies and to indicate the maximal parallelism of nested loops. Beletskyy et al [3] propose an approach to parallelize loops with nonuniform dependences. By adopting a hyperplane based representation, they present how transformation matrices can be found and applied to loops with both uniform and non-uniform dependences. Ricci [16] proposes an abstract interpretation to parallelize loop nests. Using this abstract interpretation, the author aims to reduce the complexity of the analyses needed for loop parallelization. Lim et al [13] use an affine partitioning framework which unifies different transformations to maximize parallelism while minimizing communication. In the proposed approach, they find the optimal affine partition that minimize communication and synchronization. The only other constraint network related compiler work we are aware of are [6] and [7]. These studies however focus on optimizing data locality in single processor based systems rather than enhacing parallelism in chip multiprocessors. Therefore, they are in a sense complementary to the approach described in this paper.

6. CONCLUSIONS AND ONGOING WORK

Traditional code parallelization strategies may not be the best option for chip multiprocessors where off-chip data accesses are costlier than on-chip interprocessor communication. This paper studies the problem of code parallelization for chip multiprocessors and show that a constraint network based approach can be a promising option, since it helps us maximize the data sharing among the different loop nests of an application. This paper explains how the code parallelization problem can be cast as a constraint network and how the resulting network can be searched for acceptable solutions. The paper also presents an experimental evaluation of this constraint network based approach and compares it quantitatively to three alternate code parallelization schemes. The results obtained from our implementation show that not only a constraint network based approach is a viable one (since its solution times are reasonable) but it is also a desirable one (since it outperforms all the other three parallelization schemes). Our ongoing work includes integrating this approach into a comprehensive optimizer for chip multiprocessors.

7. REFERENCES

[1] M. Arenaz, J. Tourino, and R. Doallo. A gsa-based compiler infrastructure to extract parallelism from complex loops. In *Proc. of the 17th Annual International Conference on Supercomputing*, pages 193–204, 2003.

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

[3] V. Beletskyy, R. Drazkowski, and M. Liersz. An approach to parallelizing non-uniform loops with the omega calculator. In *Proc. of the International Conference on Parallel Computing in Electrical Engineering*, 2002.

[4] K. Bondalapati. Parallelizing dsp nested loops on reconfigurable architectures using data context switching. In *Proc. of the 38th Design Automation Conference*, pages 273–276, 2001.

[5] W.-L. Chang, C.-P. Chu, and M. Ho. Exploitation of parallelism to nested loops with dependence cycles. *Journal on System Architecture*, 50(12):729–742, 2004.

[6] G. Chen, M. Kandemir, and M. Karakoy. A constraint network based approach to memory layout optimization. In *Proc. of the Conference on Design, Automation and Test in Europe*, pages 1156–1161, 2005.

[7] G. Chen, O. Ozturk, M. Kandemir, and I. Kolcu. Integrating loop and data optimizations for locality within a constraint network based framework. In *Proc. of International Conference on Computer-Aided Design*, 2005.

[8] R. Dechter. *Constraint Processing*. Morgan Kaufmann Publishers Inc., 2003.

[9] R. Dechter and J. Pearl. Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34(1):1–38, 1987.

[10] G. Goumas, N. Drosinos, M. Athanasaki, and N. Koziris. Automatic parallel code generation for tiled nested loops. In *Proc. of the ACM Symposium on Applied Computing*, pages 1412–1419, 2004.

[11] K. Hogstedt, L. Carter, and J. Ferrante. On the parallel execution time of tiled loops. *IEEE Transactions on Parallel Distributed Systems*, 14(3):307–321, 2003.

[12] I. Kadayif, M. Kandemir, and M. Karakoy. An energy saving strategy based on adaptive loop parallelization. In *Proc. of the 39th Design Automation Conference*, pages 195–200, 2002.

[13] A. W. Lim, G. I. Cheong, and M. S. Lam. An affine partitioning algorithm to maximize parallelism and minimize communication. In *Proc. of the 13th International Conference on Supercomputing*, pages 228–237, 1999.

[14] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. In *Proc. of the Conference on Design, Automation and Test in Europe*, pages 10296–10301, 2003.

[15] A. Navarro, E. Zapata, and D. Padua. Compiler techniques for the distribution of data and computation. *IEEE Transactions on Parallel Distributed Systems*, 14(6):545–562, 2003.

[16] L. Ricci. Automatic loop parallelization: an abstract interpretation approach. In *Proc. of the International Conference on Parallel Computing in Electrical Engineering*, pages 112–118, 2002.

[17] E. Tsang. A glimpse of constraint satisfaction. *Artificial Intelligence Review*, 13(3):215–227, 1999.

[18] Y. Yu and E. H. D'Hollander. Loop parallelization using the 3d iteration space visualizer. *Journal of Visual Languages and Computing*, 12(2):163–181, 2001.