Software Support Memory Forwarding

Rong Yan, Minglong Shao
Computer Science Department
Carnegie Mellon University
{shaoml, yanrong}@cs.cmu.edu

Abstract:

Memory forwarding is an effective way to dynamically optimize the data layout. It provides a safe way to improve the performance of cache by actively creating better data locality. The existing memory forwarding mechanism needs additional hardware support to perform the address redirection, which greatly limits its real application. To put the memory forwarding technique into widely practice, we propose a new way of software-based implementation. Based on the fact that good data layout can bring much improvement of cache performance and actual forwarding is a rare event during the execution, our experiment shows that reasonable speedup that can be achieved by using this software-based memory forwarding technique.

1 Introduction

1.1 Overview

With the growing gap between the speed of memory and processor, the memory latency will be more likely to dominate the execution time in most of our modern processors, especially when running some memory access intense applications, such as database transactions, and multimedia applications. While cache appears to be a reasonable solution toward addressing these problems, it still has many limitations, especially in those applications with irregular memory access pattern where the high cache miss rate invalidates the gains of cache-memory hierarchy. While the technique of prefetching can hide the cache miss, it works well only in the cases when the data addresses are predictable and memory bandwidth is sufficient.

In order to reduce the cache misses and facilitate other cache performance-improvement mechanisms as well, one possible way is arranging data layout regularly. There are two possible ways to manipulate the data layout: static placement and dynamic arrangement (also called data relocation). Data relocation is more preferable because it can adapt to the program behavior. [1] gives three steps of general relocation-based data layout optimization: guaranteeing correctness, estimating the cost/benefit tradeoff, and generating relocation code.

Targeting ameliorating the spatial property of irregular data layout at run time, memory forwarding technique [1] can safely rearrange the irregular memory blocks to a contiguous memory space at runtime. The basic idea behind this is that when relocating an object to a new address, we store the new address in the old address and mark the old location as a forwarding address. The key point here is that some hardware or software mechanisms must be applied to recognize the forwarding addresses and automatically forward the reference to the new addresses, thereby guarantee the correct result. Noting the fact that non-relocated addresses are far more common cases in the memory accesses and references to the old addresses are updated to new locations, the forwarding action is a rare event during the execution, thereby the overhead of data

layout manipulation is likely to be amortized by the improvement of cache performance.

In effect, memory-forwarding mechanism efficiently builds up a memory indirection level to increase spatial locality, and provide safety-protection mechanism to keep correctness of original memory access behavior.

1.2 Related Works

The idea of memory forwarding is closed related to those studies occurred in the context of architectures that directly supported the Lisp programming environment [2, 3]. But there are some essential differences between the two techniques with respect to the motivation, objective, implementation method, and application. The detailed comparison between the two can be found in the [1].

The implementation of memory forwarding mentioned in [1] is built on the extension of memory structure plus some software support. It assumes a new memory structure of an extra forwarding bit per 64-bit memory word. The forwarding bit serves as the flag identifying the forwarding address. Whenever memory access finds that the forwarding bit is set, the reference will be relocated to the new location indicated by the content of the old address. Figure 1 is taken from [1] illustrating an example of data relocation with memory forwarding.

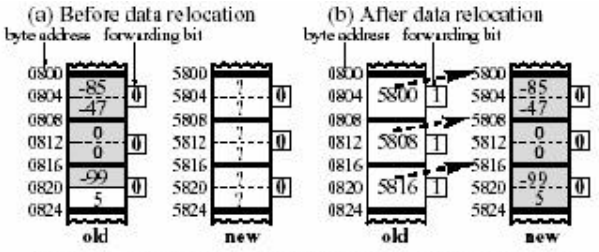

Figure 1: Memory forwarding with hardware support

1.3 Forwarding without hardware support

Hardware-supported memory forwarding showed an impressive speedup in most applications by effectively optimizing the data layout with relatively small overheads. But the assumption of underlying memory structure limited its application and the experimental results are obtained on the simulator. Why not consider to realize it through software way? It is supposed to have more flexibility and easy to be applied on the existing systems.

Intuitively, the software-based memory forwarding mechanism should guarantee all of its implementation in "pure" software, without any additional hardware support. The impact of excluding the hardware support is twofold: once the additional hardware support is unavoidable, it will hinder further application in practice and narrow the research of memory forwarding mechanism in simulator study, instead of a real machine. In contrast, software-based mechanism can be widely applied in different hardware platform, which will make forwarding technique available to current hardware architecture.

Compared software-based solution to the hardware-based solution, the essential difference is that the software supported memory forwarding cannot assume the forwarding bit of each memory word. Hence it must apply other mechanisms to identify the forwarding addresses. Actually software implementation needs to simulate the forwarding bit in hardware-based mechanism. To do this, two approaches come to our mind. One approach is to setting page protection. This approach acts like the virtual memory system. All the memory pages containing forwarding address has to be marked as "invalidate", and when these "invalidate" page is access by user program, user program will generate the signal of "page fault" and call exceptional routine to

handle forwarding address by looking up mapping table. However this approach is likely to have large false sharing overhead since the granularity of page seems to be too large for the memory forwarding implementation.

Another approach involves storing and testing for a certain magic number in place of a forwarding pointer. The magic number serves as the forwarding flag. Once user program accesses the memory occupied by the magic number, it will try to look up forwarding address in a mapping table for the final address based on the accessed memory address.

But since software-base forwarding have to insert safety-protection before each memory reference operation for the pointer forwarding correctness, it also necessarily results in significant extra software overhead, which turns to be a marginal cost in hardware-base mechanism. The additional safe-protection operations per memory access may cancel potential gains derived from good data layout. It is clear that we have to put more concern on the cost/benefit tradeoff than we did with hardware support, as will be discussed further in section 2.

1.4 Contribution of our study

Our study makes several contributions as follows,

1. We propose a software-based memory forwarding mechanism without any hardware support, which makes memory-forwarding technique available for current machines

2. We developed an automatic memory-reference profiling tool and provide user program interface to reduce the difficulty for user to incorporate memory forwarding technique.

3. We prove the user program can really benefit from our memory forwarding optimization by quantitatively evaluate benefits after applying the technique in a set of non-numeric applications.

In the following section, we will discuss in detail about the different methods in our experiment with the consideration of cost/benefit tradeoff. The rest of this report is organized as follows. We present the basic idea and performance analysis of software based memory-forwarding mechanism in section 2. Some design issues related to the cost/benefit tradeoff are discussed in section 3. Section 4 will discuss the specific implementation. Section 5 shows our experimental results. We conclude and indicate the possible further directions of research in Section 6.

2 Software-based Memory Forwarding Mechanism

In this section, we briefly present our basic idea of software-based memory forwarding, introducing some basic concepts used in following sections. We also discuss a number of performance issues in software implementation, especially considerable additional instruction overhead caused by software instrumentation.

2.1 Basic Idea

As mentioned in section 1, we will implement the software-based mechanism by storing and checking the magic number in forwarding address (Note: Forwarding address refers to the source address that already forwarded to a new destination address). The old data address is forwarded to a “memory pool” that pre-allocated by program, and the content of old data address are all storing magic number after relocation. In this sense, the magic number is used as forwarding “mark” to

identify which address is supposed to be forwarded to a new address. Although all the forwarding address supposedly store the magic number, we should prevent the unexpected case in which some unforwarding addresses happens to store the magic number and mislead the system to invoke the forwarding mechanism. Thereby, an additional address-mapping table must be managed to keep track of the forwarding mapping relation to guarantee the safe data relocation, just like the virtual-physical address mapping table used in the virtual memory system. When program believe it is dereferencing forwarding address for checking out magic number storing in that address, it will map the source address to the destination address in the mapping table. Finally, the corresponding content of destination address is sent back to the program. Typically mapping table includes two fields, e.g., source forwarding address, and destination address. When necessary, an extra field of forwarding size can be involved to provide more safe protection. At first glance, the space and time overhead of mapping table maintenance is only a trivial portion of whole overhead, but our experiment indicate the cost of inserting and updating this mapping table couldn't be overlooked in the performance, as will be discussed in section 2.2.

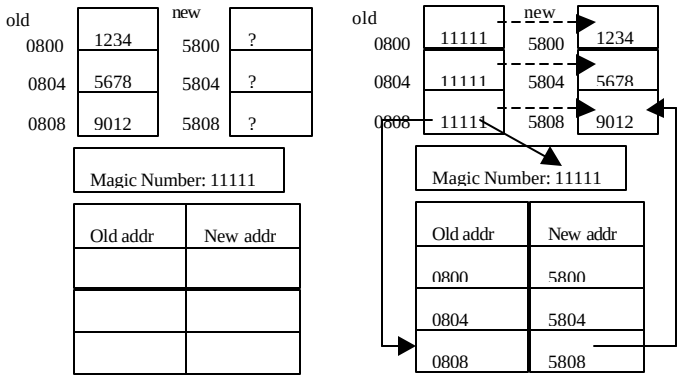

Figure 2: Example of data relocation with software memory forwarding

Before taking a further step, we will present several implications in terms of memory storage. Two implications in hardware-based mechanism [1] do still take effect in our implementation:

(i) Minimal data relocation unit is the length of a pointer, which called a word;

(ii) Performing byte-size memory access of forwarding address is allowed, but byte offset of new location should be the same to the original address;

Along with above implication, our software-based mechanism has its specific implications. First, rather than forwarding the single word one-by-one, our study shows that practical program is more likely to forward its basic data structure including a bunch of words together, and thereby we assume the memory space in a basic data structure is continuous whenever before forwarding or after. The concept of basic structure, which we refer to as a "unit", can help us reduce most of the space cost in our mapping table. Second, the length of magic number must be less than the word length, since it would be able to store in a word of forwarding address. But it is not necessarily restrict the magic number do occupy all the word space, which give us much more flexibility to capitalize on the magic number. Finally, in our implementation, only the pointer-related operations are going to be protected, for the reason that the scalar is not supposed to be forwarded and scalar-related operation would never access the dynamically allocation space. These implications would be effective throughout our whole paper.

2.2 Forwarding Gain VS Overhead

It is our main concerns that whether the performance after our optimization can efficiently improve our program performance or not. Although the software-based mechanism introduce much more unavoidable overhead than hardware-based one, once the gains are able to defeat the overall overhead, it comes the win. We now describe some potential gain and overhead.

2.2.1 Gain

Similar to the hardware-based mechanism, the potential gain of memory forwarding comes from the restructuring data layout, which enable new layout to mitigate the problem of memory latency and thereby enhance the cache performance. In brief, to restructure the data layout tends to compact the data closer, achieves more compact data layout, and does improve the spatial locality. Also memory forwarding can make best use of limited memory bandwidth, increase the prefetching effectiveness, and reduce false sharing in cache-coherent shared-memory multiprocessing system.

However, it is not always true that the new data layout can outperform the old one. Therefore programmer should ensure that the optimization of data layout can actually achieve better memory reference performance, even without consideration of any software overhead. If this condition holds, memory forwarding is worth considering.

To this end, concern should be given to figure out when memory forwarding can provide better data layout, which overcomes additional software overhead. In particular, when meet the case in which data layout is irregular, data that often visited in a row however are sparsely allocated, which unable to exploit the benefit of whole cache line, it comes our time to optimize by memory forwarding. Basically, some dynamically allocated data structures, such as linked list, tree and hash table, will be major optimization sources.

2.2.2 Overhead

The overhead of software-based mechanism grows larger than the hardware-based one. We will describe this as follows,

i. Forwarding overhead: This overhead refers to as the overhead caused by the forwarding the source blocks to the destination blocks. It's the main and almost exclusive overhead in the hardware-based implementation, but our experiments shows that this overhead only constitutes a minor portion of our overall overhead, which is not the major concern.

ii. Table maintenance: Surprisingly, the overhead introduced by the table maintenance overhead is much greater than forwarding overhead. How to reduce table maintenance overhead becomes an unexpected problem beyond our initial concern. The overall performance is so considerable due to these two reasons:

iii. 1. The mapping table needs allocation of larger memory space than forwarding memory pool. Supposedly, if table represents the mapping relationship in the granularity of word, put in another way, each table items occupying three words describe information of a forwarding word, leading to three times large space to the forwarding memory pool. This

will greatly limit the availability of memory forwarding technique. 2. table insertion time and deletion time tends to be larger than the time for forwarding memory block, and unavoidably introduce unexpected cache miss shown in our experiment, offsetting the potential gain by memory forwarding.

iv. Safe-protection instrumentation: To provide safe memory access when program happens to access the forwarding address, we have to instrument safe-protection procedure before each load/store instruction. Although the forwarding process is seldom invoked and the forwarding look up time is negligible in most of applications, the critical path of program grows two or three times longer by additional comparison and conditional branch instructions. Since these new instrumentation instructions depend on previous instruction, the potential parallelism is hurt dramatically and the benefit of pipeline cannot be fully exploited.

v. Additional space overhead: memory forwarding optimization called for several times larger memory space than original program, which prevent the program from handling original maximum number of data set.

In essence, the memory forwarding optimization can outperform only if the overall performance gain could outweigh the extra management overhead. Moreover, the table maintenance and safe-protection instrumentation cost make up most portion of overall overhead. We will discuss further on how to reduce the overhead in our implementation.

3 Design Issue

Before providing further insight on the implementation issues, several fundamental design principles should be clarified first.

3.1 Basic Design Principles

In this section, we discuss the basic and critical design principles of memory forwarding implementation.

3.1.1 When to insert safety-protection procedure

Generally speaking, we should take care of each pointer-related operations, that is, every operations using pointer-type variables as operands should be protected by additional safe-protection procedure. All the operations have to " interpret" the initial address to the final address before it is taken into effect, as defined in [1]. Mostly, the pointer-related operation is to dereference the pointer and obtain the memory content. Another easily oblivious case is when pointer is treated as normal integer and related operations are performed on it, such as pointer comparison. In some cases, it is tough for compilers to interpret pointer to final address when the pointer is forcedly converted to another type. For instance, if someone builds up a hash table with address as its key, the address has to be changed to integer type before looked up in the hash table. After memory forwarding, the compiler is more likely to overlook this case since the pointer has already be converted to non-pointer type and not yet to be considered.

3.1.2 How to choose the magic number

Based on the fact that we want to use a magic value to replace the forwarding bit and identify a forwarding address. This value should not be used (or used only very rarely) in the original execution. Intuitively, we must find such a value across different applications, which is possible given the huge range of 32-bit values. In practice, the magic number is randomly chosen whereas cause little false forwarding, and this is proven to be a trivial thing in our implementation.

However, it is interesting to investigate different representations of magic number, for the reason that magic number can carry additional information for special purposes. We propose four types of magic number representation below

1. " Pure" Magic Number (32 bits). The whole word space was used to store magic number, without any other information;

2. Magic Number(24 bits) + Offset(8 bits). The offset is referred to as the word offset in the forwarding data unit. When using the forwarding table on the granularity of data unit, only the starting address is stored in the forwarding table. The information of offset is necessary to align the arbitrary address to the unit starting address.

3. Magic Number(8 bits) + Table Index(20 bits). The table index is referred to as the item index in the mapping table. So program can quickly find the corresponding table item without any support from hash table.

4. Magic Number(12 bits) + Partial Table Index(16 bits) + Offset (4 bits). Here comes the most complex representation of magic number, which is trade-off representation between time and space consumption. We partition the word into different parts, one stands for the flag of forwarding address (the original meaning of magic number), one stands for the highest 16 bits of table index in the linear index table, and one stands for the offset as mentioned in former case.

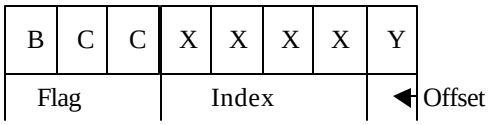

Figure3 Representation 4 for Magic Number

3.1.3 How to return the memory content of forwarding address

Basically, there are two schemes to return the corresponding content when the program encounters the memory forwarding address. The first intuitive scheme is shown in Figure 4a, when initial address is found to store a magic number, the software step into looking up the final address, then dereference the final address, and return the content of final address to destination register. Software has to iteratively interpret the initial address into final address unless it makes sure that final address is no longer a forwarding address. But main problem of this scheme is unable to apply for store into forwarding pointer, since store operation

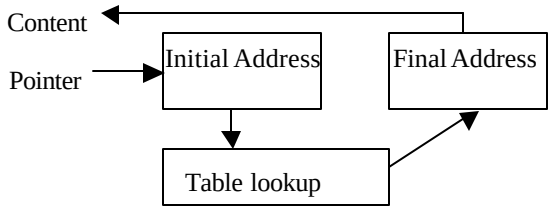

Figure 4a

only needs information of the final address, rather than its content.

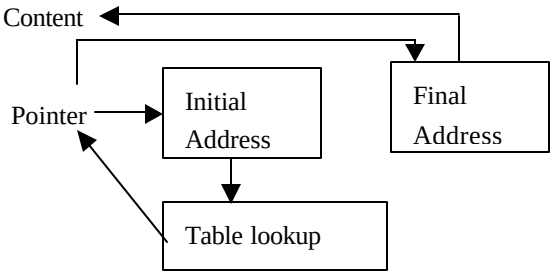

Figure 4b

The second scheme is trying to redirect the pointer to be the final address after looking up the mapping table, and access the new memory space with this redirected pointer. This scheme permanently redirects the pointer into final address, thereby avoiding the future abundant redirection and optimizes away the future forwarding on the fly. More importantly, this scheme can be applied in store operation without any difficulty. So we will use latter scheme in practice.

3.2 Step-by-step issues

Having discussed the basic principles of memory forwarding design, we now focus on the issues on each memory forwarding step.

3.2.1 Initialization

The memory pool and mapping table should be initialized in initialization step. In our implementation, we allocate two separate and large chunk of memory block to serve as the memory pool and mapping table, while mapping table cannot be directly accessed by user application. To guarantee the forwarding efficiency, the memory pool must be large enough to hold the relocation data

3.2.2 Relocation

The purpose of this step is to forward the old data into memory pool, and to update the mapping table. When the maximum forwarding capacity of memory pool is reached, the memory pool will simply refused to accept new relocation data. As earlier analysis, the table maintenance overhead should be a major concern. Hash table and linear has different impact on the relocation performance, as will be discussed in next section.

Another interesting issue is whether the data can be relocated once or relocated multiple times? Multiple time relocation can improve performance in some case, such as the frequent insertion/deletion of data structure, but it leads to large maintenance cost and difficult deallocation strategy. In contrast, one-time data relocation can save a large number of comparison instructions and conditional branch instructions by not tracing along the forwarding chain, and reducing some unnecessary relocation time. We can implement this by canceling the relocation operations when it found the memory block has already been forwarded. As shown in our experiment, this simplified scheme can work well in some applications.

3.2.3 Free

Free step is an "annoying", and greatly related to the relocation step. The typical implementation of free step is that when program intended to free a memory space, first check if current address is forwarding address, if so all the related address in the forwarding chain might be deallocated at the same time. But drawback of this scheme is unaffordable time consumption and requirement for additional mapping table support, in order to search backwards from final address to initial address. The one-time reallocation scheme mitigates the program to some extent since the address can only once forwarded, but the backward mapping table is still unavoidable. For the purpose of performance, we can take a step back, limited the program not allowing to release the address in a forwarding chain, including the initial address and final address before the final clean-up stage. All the forwarding address will be deallocated together at the clean-up step.

3.2.4 Clean-up

Clean-up step aims at cleaning the entire additional space that is unable to handle in the user program, including the memory pool and the mapping table.

3.3 Additional Concern

i. Compared with the hardware-based implementation, software-based mechanism has its own characteristic. First no extension of additional instruction set is required. The additional unforwarded memory operations in hardware mechanism can simply make use of the raw load/store instruction without the safety-protection instruction in our mechanism, and the normal load/store instruction would be protected by the forwarding safe-protection code. Second, the data dependence speculation will be a great help to our performance, since we have introduced much more branch instruction with the safe protection code, but the forwarding is almost never happened. We can expect the data dependence speculation technique will improve the forwarding performance a lot. Finally, software-based mechanism is much simpler to profile forwarding-related information without providing user-level trap, such as counter of forwarding load/store.

ii. Avoid calling in new source of cache miss simultaneously when reduce the cache miss by data relocation

iii. Although optimizing an existing program by forwarding technique require the application-specific knowledge, forwarding optimization should not mess up the readability of program. It might provide a set of program interface for user to enjoy inserting additional optimization code and profiling tool to automatically insert the additional safe-protection code.

4 Implementation

In this section, we present some implementation structure and further discuss several possible implementation details.

4.1 Structure

Basically, the software based memory forwarding is composed of two levels: user application interface and compiler-level support. Its structure is showed below (Figure 6):

The user interfaces provide APIs for the user application to relocate addresses and dynamically optimize the data layout of the program. The compiler support takes the responsibility to automatically forward old addresses to new addresses in a way that is transparent to the application programmer, which won't cause extra burden of programming involved in guaranteeing the correctness of memory accesses. And implementing the frequently-called checking code at a low level is also the requirement of performance.

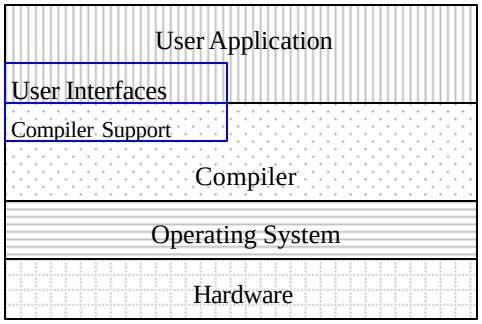

Figure 5 software based memory forwarding

Compiler Support: As we mentioned earlier, each pointer-related memory access will be expended to several related steps: loading the content of the address, comparing it with magic number, looking it up in the table if match and updating the pointer if it points to a forwarding address. These additional operations are very expensive and we should reduce the overhead as much as possible. SUIF[4, 5, 6] is feasible way to insert extra "instructions" before each any instructions in the user programs. Along with the aid of SUIF, DDAN is a useful profiling pass to collect the profiling information of each memory reference instruction. Although the initial objective of DDAN is used to collect the profiling data and not get much concern with the performance of final code, it provide a way for us to implement automatically instrumentation. We modify the DDAN profile pass insert memory forwarding safety-protection code to meet our goal.

User Interface: We provide four APIs for user program, corresponding to each steps discussed in section 3.2:

mf-initialize(); Setup the necessary environment for user application. It includes the creating of address mapping table(s), and initializing a memory space serving as destination addresses which holds the real data.

*Relocate(UINT **src, UINT n_word); src* refers to as the initial address that users want tot relocate. n_word is number of Relocate the original address, updating the address mapping table, storing a magic number, moving the data to the new space, and updating the pointer to the new address. Relocate function is the most frequently called function and its implementation will affect the whole performance of software based memory forwarding. Several versions of tables and relocate methods are tried in the experiments to compare the different performance and learn some general behaviors in the memory forwarding.

*myfree(UINT *src):* Our own deallocation interface to release the address *src*, whether it is forwarding address or not. Normally, this function is invisible to the user program, and the

standard free function will be automatically replaced by it.

mf-finish(); Free memory space and address mapping table(s).

Programmer Intervention: Software based memory forwarding is a useful technique, but it is too costly if inserting safety-protection code to all the instructions. To avoiding abundant safety-protection overheads, extending only pointer related memory access is first step towards this direction. But to further optimize away the unnecessary safety protection code, we need most of the application-specific knowledge thereby it is difficult for compiler to automatically figure out where the safety protection code is exactly indispensable.

In this point, we want to investigate when the safety-protection procedure is really useful in our application. If the programmer hopefully ensures all the current reference to a forwarding address has already been removed, we guarantee this memory space will not to be accessed again. Therefore the future safety-protection code is useless to this address. This is referred to as the perfect memory forwarding. While the case is not achievable, it gives us an idea that we only need to instrument some additional protection code in some regions we believe to be dangerous.

To this end, we provide some simple interfaces for programmer to indicate compiler which "dangerous" part of codes need special attention. In our experiment, we compare the programmer-intervention method and compiler-only method. And the experimental results support this conclusion. How to make compiler more intelligent to automatically find out the "dangerous" is an open question to future study.

4.2 Several Different Implementations

To obtain the good performance as much as possible and to understand the general behavior of memory forwarding, we tried several different solutions.

Address Mapping Table: What kind of table is most desirable? We tried hash table and simple linear list. At first, we expected that the hash table will give better performance, but it turned out to be worse than a linear list. The reason for that is looking up in the table is a rare event but relocation can be a frequent operation, which means a table with simple insert operation is much preferable to a table with simple retrieval operation. Attaching a new element at the end of list is much simple and the retrieval also won't become bottle neck if we designed the magic number carefully. Another important reason is, using hash table, we didn't eliminate the irregular data layout, but just transfer it from the user program to our operations. Take the simple and common hash function: index = address % table_size for an example. Because the table_size is larger than the distance of two addresses, after the "mod" operation, the distance of index for two addresses remain the same. Those irregular data layout in the user program will be transferred to our address mapping table!

Eliminate Table Maintenance: What if we haven't the address mapping table at all? In the experiments, reference to the old address are rarely happened if not never. If all the pointers can be guaranteed to have the new value, we may not maintain an expensive table at all. Actually, this violates the correctness requirement of memory forwarding, but we can use it to get an upper

bound performance.

The Granularity of Forwarding: The granularity of a word is not feasible in the software based memory forwarding as it was in the hardware-supported case. Too fine granularity means long mapping table and more checking and looking up operations. We implemented both word granularity and structure granularity in our experiment to justify this.

In summary, we have tried about 20 implementations of each test program in order to get the general and accurate behavior of software memory forwarding.

5 Experimental Results

We choose two non-numeric benchmarks and one our own tree traverse program in the experiment. We also manually apply some relocation-based optimizations on these programs. The goals of the optimizations are improving special locality and reduce cache miss rate. Following is the application characteristics and experimental environment.

Table1 Application Characteristics

Name	Description	Source	Default Data Set	Optimizations Applied	Instructions Completed
Heath	Simulation of the Columbian health care system [8]	Olden	max.level = 5/4 max.time=500/1000	List linearization	200M
MST	Finds the minimum spanning tree of a graph [8]	Olden	1K node	List linearization	414M
Tree	Simple tree traverse	Ours	2-folds, 20 levels	Subtree Clustering	11M

The experiment platform is Linux 2.4.7 (kernel modified by PAPI [7] for the testing purpose) on Intel Pentium III (1GHz). We also collect some data on the OSF1 on Alpha machine.

Table2 Characteristics of Caches in Intel Pentium III [9]

Cache or Buffer	Characteristics
L1 Instruction Cache	16Kbytes, 4-way set associative, 32-byte cache line size
L1 Data Cache	16Kbytes, 4-way set associative, 32-byte cache line size
L2 Unified Cache	256Kbytes, 4-way set associative, 32-byte cache line size

To fully investigate the impact of the different software-based implementations, we have implemented several different versions of software-based mechanism. Partial major testing version is shown as below,

Table3 Different Implementation

Version	Table Type	Granularity	Magic Number	Comment
INDEX_WORD	Linear	Word	MAGIC + Index	
HASH_WORD	Hash	Word	"pure" MAGIC	Two hash table Multiple Relocation
HASH_UNIT	Hash	Unit	MAGIC + Offset	Two hash table Multiple relocation
HASH_UNIT_SHORT	Hash	Unit	MAGIC + Offset	Two hash table

				Relocate one-time
HASH_ONE_TABLE	Hash	Unit	MAGIC + Offset	One hash table
LINE_TABLE	Linear	Unit	MAGIC + Partial Index + Offset	

For each benchmark application and different memory forwarding implementation, we will compare the original program performance (denoted Original) and performance after forwarding optimization (with DDAN). In addition to these cases with and without forwarding optimization, we also show a case with perfect memory forwarding (Perf). In this case, we assume all the references to relocated memory space access them in the new address, and we cancel all the safe-protection code in the program. Although it is unsafe without forwarding protection, this performance shows the upper boundary of our real forwarding optimization. In particular, we also shows the performance of our manually instrumentation case (MANUAL) in health problem, as presented in the section 4.1. It can reduce lots of unnecessary safety-protection code as long as user believe the "exposed" part is safe in the future memory access, meanwhile it reduces large potion of overhead in the applications, and leads to a reasonable speedup in health program.

5.1 Performance of Memory Forwarding Optimization

We now illustrate the testing results on the Linux platform. Here we start with analysis of the performance of memory forwarding optimization (With DDAN), perfect memory forwarding (Perfect) and original program (Original). All the results below are based on LINE_TABLE implementation as denoted above. In each figure below, the three clustered bars are referred to as the health benchmark (health), MST benchmark (MST), and tree traverse (TREE) from left to right. Each bar in Figure 6 represents the total normalization execution cycles. The Figure 7 shows the L1 data cache miss times. Figure 8 represents the number of completed instruction. Figure 9 shows the number of total used conditional branches.

Our first observation from figure 6a is that software-based forwarding mechanism does achieve reasonable speedup in MST and TREE application. But the total cycles do increase a lot in the HEALTH benchmark. This result is greatly related to the specific data structure, frequency of referencing address and relocation time of each application. MST optimize the data structure in the beginning by relocate the hash table into compact space, and not so often dereference the address in its post-relocation computation time. So the software-based mechanism can enjoy increasing data spatiality without introducing large portion of safety-protection overhead. Nevertheless, Health always applies insertion and deletion operation in the linked list and memory reference operation makes up a major part of execution time. We have to optimize the linked list from time to time throughout the processing time and therefore the program suffers from the great amount of additional code and overhead.

The perfect memory forwarding always outperforms the original problem performance in these three cases, due to the better data layout and increased spatial locality (But the perfect forwarding is not necessarily achieving better performance than the original program, as can be seen in the our data from Alpha machine).

In figure 6b, the L1 data cache misses reduce a lot in the TREE, since almost all of the continuous memory reference is not visiting the data in the same cache line. In other two programs

only achieve a moderate cache miss rate reduction, about 10%. The loss of improvement on potential cache hit rate is mainly caused by additional table maintenance cache miss and safety-protection instruction.

The conditional branch count in figure 6d gives us more insight on our software-based forwarding mechanism. The forwarding optimization with full safety-protection has inserted many conditional branch instructions, which do largely affect the performance of forwarding optimization. HEALTH is a typical example that large conditional branch cost offsets the potential gain from reduction of data cache misses. In contrast, MST suffers less than the Health benchmark in this aspect.

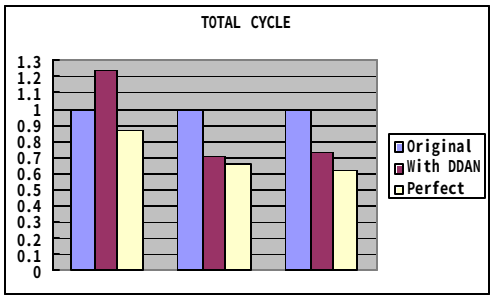

Figure 6a Total Cycles

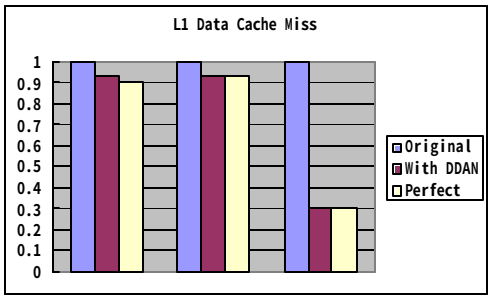

Figure 6b Cache Miss

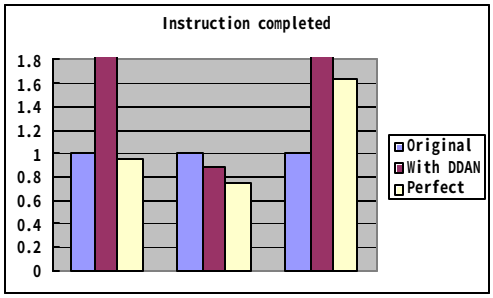

Figure 6c Instruction Completed

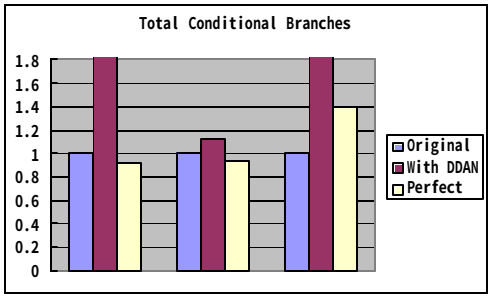

Figure 6d Conditional Branch

5.2 Performance between software-based forwarding versions

The column clusters from the left to the right stand for different software-based version of LINE_TABLE, HASH_WORD, HASH_UNIT, HASH_SHORT, and HASH_ONE_TABLE respectively. These figures clearly demonstrate the linear table gives the best performance, and hash table with word granularity is the worst among the different implementations

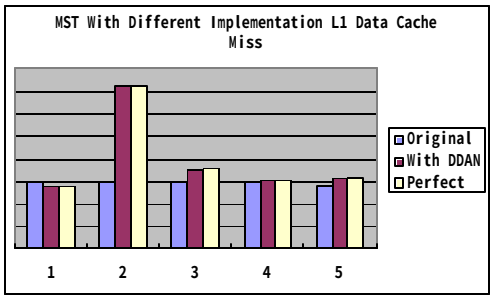

Figure 7a L1 Cache Miss in MST

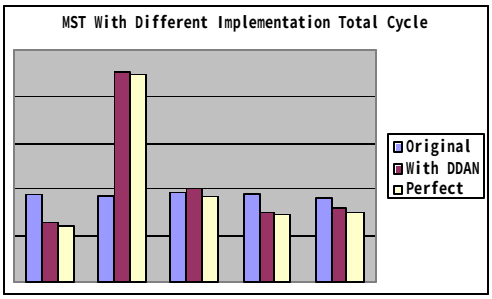

Figure 7b Total Cycle in MST

5.3 Case study on HEALTH benchmark

We now take a look at the health benchmark in more detail. Health is a health system simulator, which simulate the patient arrival and leave time to get the statistical data. Health has two arguments, denoting the number of levels of a tree and iteration times respectively. It will frequently insert and remove the element from the different linked lists, and we apply the list linearization optimization on this. An extra counter field is to keep track of how many memory operations are executed after the last memory relocation. When counter exceeds the threshold number, the program begin to linearize the linked list. The threshold is normally set to 50 in our experiment.

Figure 8a compares the effect of different threshold; small threshold means more relocation operations, thereby getting worse performance. The threshold of the first column cluster is 50, and the threshold of the second column cluster is 100. The impact of this difference is quite obvious, demonstrating the importance of threshold choice.

Figure 8b compares the effect of different arguments. Arguments for the first column cluster are 5 levels and 500 iterations; arguments for the second column cluster are 4 levels and 1000 iterations. Less tree level and more iterations leads to less data relocation times and keeps around the same amount of memory access, thus getting better performance.

We also test the two different implementations with/without table maintenance overhead. The result is given in the figure 8c. We can see that without the table maintenance, much better performance is achieved, showing that table maintenance cost constitute a large portion of overall overhead.

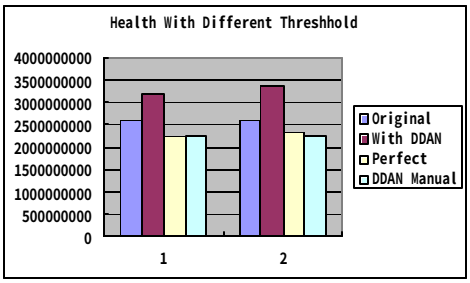

Figure 8a Health with different thresholds

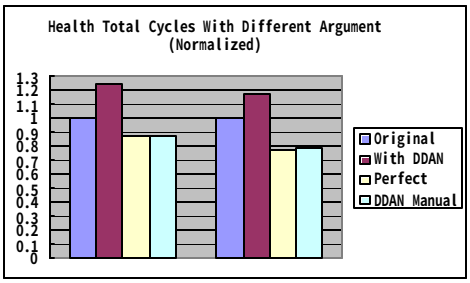

Figure 8b Health with different arguments

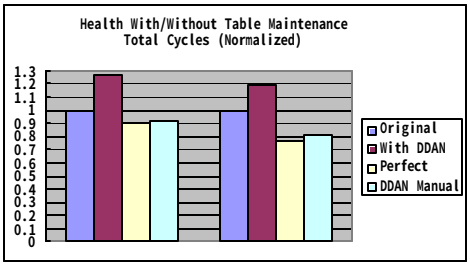

Figure 8c Health with different arguments

5.4 Performance of traversal on the basic tree structure

We now turn our attention to the study on impact of different data spatial locality. We compare the performance in terms of normalized total execution cycles. The first clustered column is the case that consequent tree structure cannot share the same cache line, and the second, on the other hand, can share the share one cache line. Observation from this figure is exactly the same from our initial imagination. When the data structure is regular, memory forwarding is unable to achieve speedup since data spatial locality only have little space to improve, but when the original data layout will come with much cache misses, it comes the time to perform memory forwarding optimization on that.

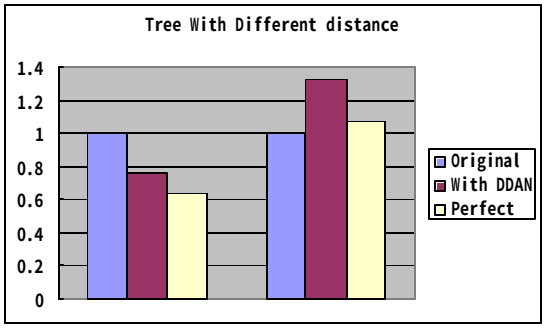

Figure 9 Health with different arguments

5.5 Performance on Alpha platform

We collect some data on the Alpha platform to see the optimization performance on different platform besides PIII. We use the health benchmark to be the test case, and each column represented different parameters in health system. Surprisingly we observe that even the performance of perfect memory forwarding is unexpectedly degraded in Alpha platform. That means the performance of our optimization mechanism is varied with different platforms.

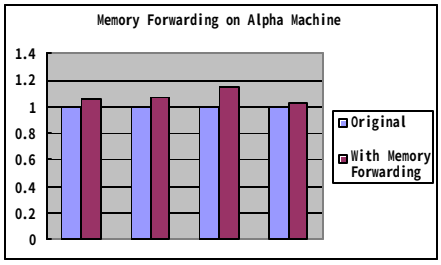

Figure 10 Memory Forwarding on Alpha

6 Conclusions

Memory forwarding is a safe way to dynamically optimize the data layout, which can improve cache performance by reducing cache misses and facilitate other cache performance improvement techniques.

The impressive performance gains achieved by hardware implementation of memory forwarding and its limitation of special hardware support motivate us to implement it through

software. Although there is a great deal of overhead in the software based memory forwarding mechanism compared with hardware-based mechanism, we also found it a feasible and promising technique. Software memory does achieve speedup on those applications especially with irregular data layout. Software mechanism also has other advantages. For instance, its flexibility allows programmer to apply it wherever necessary. We believe, with the support of operating system, the overall maintenance overhead will be reduced a lot.

In this paper, we propose some possible ways to reduce the overhead of software mechanism. Experimenting with several different parameters, we also indicate the future research efforts.

7 Acknowledgments

We would like to thank C-K Luk and Todd Morry for the valuable suggestions on memory forwarding through emails and discussions. We are also grateful to Pedro Artigas for his DDAN profile tools. And Shimin Chen gave us help information on the testing tools.

References

[1] Chi-Keung Luk, Todd C. Morry. Memory Forwarding: Enabling Aggressive Layout Optimizations by Guaranteeing the Safely of Data Relocation" . In *Proceedings of 26th Annual International Symposium on Computer Architecture* , May 1999

[2] D. A. Moon. Architecture of the symbolics 3600. In *Proceedings of 12th Annual International Symposium on Computer Architecture* , 1985

[3] G. S. Taylor, P. N. Hilfinger, J. R. Larus, D. A Patterson, and B. G. Zorn. Evaluation of the SPUR Lisp architecture. In *Proceedings of 13th Annual International Symposium on Computer Architecture*, 1986

[4] SUIF.Library.Manualhttp://suif.stanford.edu

[5] SUIF Cookbook Manual http://suif.stanford.edu

[6] SUIF BUILER Library Manual http//suif.stanford.edu

[7] PAPI. http://icl.cs.utk.edu/projects/papi

[8] M. C. Carlisle and Anne Rogers. *Software caching and computation migration in olden. In Proceedings of PPoPP'95*, pages 29–38, July1995

[9].IA-32 Intel Architecute Software Developer's Manual V3:System Programming Guide