Knapsack: A Zero-Cycle Memory Hierarchy Component*

Todd M. Austin T.N. Vijaykumar Gurindar S. Sohi

Computer Sciences Department
University of Wisconsin-Madison
1210 W. Dayton Street
Madison, WI 53706
{austin, vijay, sohi}@cs.wisc.edu

November 22, 1993

Abstract

The widening gap between processors and memory necessitates the development of novel memory hierarchies: hierarchies that can possibly service memory references at register speeds, since service at cache speeds may not be adequate. We consider the design of a novel memory hierarchy component, a *knapsack*, whose purpose is to provide (very) fast access to frequently-used data objects. Software allocates frequently-used objects into a *knapsack region* of the address space; a knapsack provides fast access (at register speeds in many cases) to these objects. We discuss how a knapsack is different from other memory hierarchy components, how it can achieve superior performance over other components, and how it can be integrated transparently into an implementation of existing architectures. We also carry out a detailed evaluation of a knapsack using several of the SPEC92 benchmark programs. Our results show that for many programs, both numeric and non-numeric, the knapsack offers significant opportunity for memory access optimization. Using a profiler-based packing heuristic which can allocate both globals and locals to the knapsack region, we found that a knapsack as small as 4k bytes could service a significant number of memory references for many of our benchmark programs.

*This work was supported by grants from the National Science Foundation (grant CCR-9303030) and Office of Naval Research (grant N00014-93-1-0465).

1 Introduction

Ever-increasing CPU speeds continue to place greater demands on memory systems. Improving memory bandwidths, and decreasing memory latencies is of paramount importance if the rate of increase in processing speeds is to be sustained. To date, most techniques to improve memory system performance have primarily used hardware-only techniques: caches[14], lockup-free caches[10][15], multiport caches[15], hardware prefetching[8] and buffers[8], etc. With increasing on-chip real estate, designers are using the additional resources to integrate larger amounts of existing memory hierarchy components on chip. Our feeling is that a brute-force integration of larger sizes of well-known memory hierarchy components may not be the best solution; other avenues need to be explored.

A trend that has emerged in the design of high-performance processors is the ever-increasing demands placed upon the software (compilers in particular). Such processors, which exploit instruction-level parallelism (ILP), benefit greatly from software assist. Without sophisticated software technology, such processors typically cannot achieve high performance levels. A natural next step is to use sophisticated software technology to optimize memory hierarchy performance. This area has received a lot of attention recently; much of this work has concentrated on improving the performance of an existing memory hierarchy component, namely the cache, using two main techniques: prefetching to reduce the number of compulsory and conflict misses [2][3][9], and blocking to increase the reuse of data once it has been cached, by changing the reference patterns of the application[11].

Our interest in this paper is not to improve the performance of well-known memory hierarchy components, but to explore the use of another component. The component we have in mind, a *knapsack*, could be accessed at register speeds, for the most part, and yet can be introduced transparently into an implementation (of an architecture). The basic idea is to map a restricted part of the address space into the knapsack using a very simple mapping strategy. The small size and restricted mapping facilitate fast access. Sophisticated software is then assigned the task of allocating frequently-used data objects in this restricted part of the address space, so that fast access to this part of the address space translates (dynamically) into fast access for a reasonable number of memory accesses.

The outline of the paper is as follows. Section 2 discusses the attributes of components found in common memory hierarchies. The purpose of this section is to discuss how certain attributes influence the use and the performance of the components. Section 3 presents the proposed memory hierarchy component, a knapsack, along with a discussion of its benefits, including potential access at register speeds. Section 4 discusses how a compiler can make use

of a knapsack, and Section 5 presents an evaluation of the knapsack using fifteen of the SPEC92 benchmark programs. Finally, Section 6 presents concluding remarks.

2 Components of Common Memory Hierarchies

A memory hierarchy is a collection of storage entities, or components, with each component consisting of several elements of storage. For purpose of discussion, we can assume that a program is referencing objects allocated in a (virtual) memory address space; objects in this address space are mapped onto components of the memory hierarchy during the execution of the program. The goal of a memory hierarchy is to service memory requests from the program with the desired latency and the desired bandwidth. A hierarchy typically accomplishes this by mapping frequently used data objects into components of the hierarchy that can service the requests faster. By mapping we mean the placement of data objects in the component of the hierarchy.

We begin by discussing the fundamental issues that characterize a component, and then discuss common memory hierarchy components. Three fundamental issues characterize how a component fits in the overall memory hierarchy:

1. Whether the component is *architecturally-visible*, and has a *separate address space*,
2. Whether the component is *addressable*, and
3. The processes of determining which component a referenced object is in, and where it is in the component.

A component is architecturally-visible if it is visible to the instruction-set architecture (ISA). That is, the ISA treats the component as a distinct part of the hierarchy. It does so by considering the component to be part of a separate address space, and providing support in the ISA to access the component. Because the component is in a separate address space and uses special addressing mode instructions, software support is *required* for *correct use* of the component. An example of a component in a separate address space is the (architectural) register file.

If a component has its own address space, software is responsible for deciding which objects reside in (or are mapped into) this component, and when they reside in the component. In order for software to map objects from one address space (the virtual address space where most[1] data objects of a program reside), into another address space (the address space of the

[1] We say most, not all, since some objects may never exist in the virtual memory space of a program. Examples

component), several problems have to be solved. One, the software has to analyze the memory references of a program to make sure that there are no aliases involving objects that it maps into a different address space. In the extreme, this boils down to determining which object an arbitrary memory operation accesses. The complexity of these analyses, in the presence of control flow and pointers, limits the number of data objects that can be mapped into a different address space. Two, since the hierarchy component is very likely much smaller than the size of all the objects that a program might want fast access to, different objects have to be mapped into the component at different times in the execution of a program. Software has to decide which objects are mapped, and when. This problem becomes very hard when the objects are of varying sizes and can't be broken up.

If a component does not have a separate address space, it is architecturally-invisible. That is, software is *not required* for *correct use* of the component. (However, architecture-invisibility *does not imply that the component is invisible to the software.* While software is not required for correct use, it can be used to optimize performance of the component.) With an architecturally-invisible component, data objects can be dynamically mapped, with the mapping process (primarily) being the responsibility of the hardware. Since software is not required to give any guarantees, extraordinary demands are not placed on it. However, hardware suffers from the lack of global knowledge of (possible) program reference patterns. Software can assist the hardware by giving it hints and advisory directives. The point worth emphasis here is that software is assisting the hardware, but is not responsible for providing it with any guarantees: the hardware can use the memory hierarchy component correctly *without* any software assistance.

The second fundamental issue deals with the addressability of the component. If elements of the component can't be addressed then it is not possible, to map objects that may be referenced indirectly into this component. (It is worth pointing out that just because a component lies in a separate address space does not mean that its elements are addressable. For example, registers form a separate address space from memory, but are not addressable if self-modifying code is prohibited.) Software analysis to determine which objects can't be referenced indirectly is hard in the presence of pointers, thereby limiting the number objects that can be mapped into this component. Related to the notion of addressability is the notion of indexing. If the component can't be addressed, it can't be indexed, that is, it is not possible to compute the address of a datum mapped into this component by adding an offset to the address of another datum. The lack of an indexing ability prevents the mapping of aggregate data objects, such as arrays, into this component. For example, since primary registers cannot be indexed in most architectures,

---

include temporaries, parameters, and local variables of a program that are resident only in the register address space during the execution of a program.

Memory Hierarchy Component	Architecture Invisibility	Addressability			Presence detection
		Indexable	Indirect	Aggregates	
Registers	No	No	No	Limited	Compile-time
Caches	Yes	Yes	Yes	Yes	Run-time
Vector Registers	No	Yes	Yes	Yes	Compile-time
Traditional Local Memory	No	Yes	Yes	Yes	Compile-time
Stack Cache	Yes	Yes	Yes	Yes	Run-time
Knapsack	Yes	Yes	Yes	Yes	Run-time

Table 1: Attributes of Common Memory Hierarchy Components

compilers cannot produce code corresponding to loops that process array elements, if the array elements are mapped into the registers.

The third fundamental issue deals with accessing data in a memory hierarchy. When a memory reference is generated, how do we know which memory component it should be directed to, and where can the referenced object be found in the component.

If the component is architecturally-visible, then the software is responsible for directing the reference to the component, and hardware is not involved in the decision making. For example, if a variable is mapped into a register, software ensures that further references to the variable are directed to the register.

If the component is architecturally-invisible, hardware has to determine which component the object is in. It does so by imposing an ordering on the components (for example L1 cache followed by L2 cache, and so on). Selected places are searched in each component, and the components are searched in order. The complexity of the search process is determined by the mapping strategy. If the mapping allows many objects from one component to be mapped into many places in another component (for example many blocks from memory being mapped into many block frames in a set associative cache), then hardware has to direct the reference to all parts of the component where the object could reside. Furthermore hardware also needs to determine which one of the several objects that could map into the selected storage elements is actually present in the component. The hardware's task is simplified if there is a many-to-one mapping (for example in a direct mapped cache), and could be simplified even further if there is a one-to-one mapping.

Table 1 presents the attributes of primary registers, caches, stack caches[6], and local memories[5]: components that are found in common memory hierarchies. Primary registers, or simply registers, are found in all modern architectures. Registers form a different address space from the memory address space, therefore objects must be mapped into registers by software. It is generally not possible to take the address of a register, and it is also not possible to index a

primary register file. These restrictions limit the number of data objects that can be mapped into registers. Typically primary registers are used to hold scalar (global) variables, temporaries, local variables and parameters.

Caches are architecturally-invisible. It is possible to take the address of a datum allocated to cache, index elements, and map aggregates. This flexibility, however, is achieved at some expense – hardware for presence detection. For example, tag bits are needed to determine which one of the many blocks that could map onto a cache block frame is actually present in the cache. Moreover, because of the general many-to-many mapping strategy, special-case optimizations are not possible.

Vector registers[13], found in vector machines, are another interesting memory hierarchy component. It is generally possible to take the address of an element of a vector register (a scalar register contains the address of the element), and use indexing to step through elements of a vector register. Thus, it is possible to allocate vectors in a vector register, and use scalar instructions (in a loop) to process elements of a structure mapped into a vector register. However, because vector registers form a separate address space, only data objects whose complete alias information is available can be mapped into vector registers.

Very few architectures have local memories; only the Cray-2[5] comes to mind. The Cray-2 local memory is conceptually very similar to a vector register (that is, it has its own address space), only much larger. The one difference is that, because of its size, it can hold many different, unrelated data objects at a time, unlike a vector register which typically holds part of one object. This extra flexibility, however, opens up additional problems: the issue of packing multiple, variable-sized objects into a fixed amount of space. Because of these issues, the primary use of the Cray-2 local memory has been a place to spill vector registers.

A stack cache, proposed for the C machine, is another interesting memory hierarchy component[6]. The goal of the stack cache is to provide fast access to local objects and parameters of procedures – objects that reside on the top of a run-time stack (for the typical execution of block-structured languages). By prohibiting modifications to the stack pointer during the execution of a procedure, by providing special instructions to adjust pointers at procedure entries and exits, and by predecoding instructions to determine if their operands are mapped into region of memory currently held by the stack cache, the stack cache allows top-of-stack elements to be accessed at register speeds. Even though the stack cache does not have a separate address space, it is not completely invisible to the software. Special instructions (*enter* and *catch*) are needed to adjust pointers into the stack cache[2]. Furthermore, a stack cache holds *all* local variables of

[2] One way of looking at the software requirement is as follows. Since the stack cache could hold a region

a procedure, rather than only variables that have suitable access characteristics (for example, high access frequencies).

To summarize the above discussion, there are three desirable attributes of a memory component. First, the component should be architecturally-invisible: software should not be *required* for *correct use*. This also implies that it does not have a separate address space, and can be addressed and indexed. Second, software should be able to *assist* in performance aspects. Third, run-time decision making, regarding where a referenced memory object is present, should be as simple as possible.

The next section describes a new memory hierarchy component, a *knapsack*, that we propose and study in this paper. The goal of the knapsack is to achieve the same purpose as a very large addressable and indexable register file (fast access), but be architecturally-invisible so that software is not required for *correct use*, and therefore it can be used in implementations of existing architectures. We accomplish these goals by restricting the mapping of memory locations into the knapsack, thereby simplifying the hit detection process (and allowing fast access), and relying on software to map frequently-accessed memory objects into the knapsack.

3 A Knapsack

3.1 Basic Idea

The idea of a knapsack[3] , is very simple: provide fast access to a restricted part of the memory address space (using a knapsack), and use a one-to-one mapping strategy to simplify the hardware decision-making process. The basic idea of fast access to a part of an address space in not new – it has been used many times before. The novelty of the work presented in this paper is profitably integrating this simple idea into the memory hierarchy of a modern processor. Software is not essential to the use or to the correct operation of this component, but software can (and should) be used to optimize its performance. Figures 1a) and 1b) illustrate the concept of a knapsack. Without a knapsack, generally an element residing in any part of the address space can reside in a memory hierarchy component, for example, in a cache. With a knapsack, the memory address space is divided into two regions: X and Y. The knapsack is responsible solely for providing fast access to region X, also called the *knapsack region*; other

of memory that is much larger than it size, at different times during the execution of a program, software is required to ensure that only a fixed-size region of memory (and everything from that fixed-size region of memory) is mapped into a stack cache, at a given time.

[3]The name comes from the fact that the allocation of variables to this memory component is similar to the Knapsack Problem[4]. Moreover, a knapsack serves as a repository for objects that are "valuable" to a program.

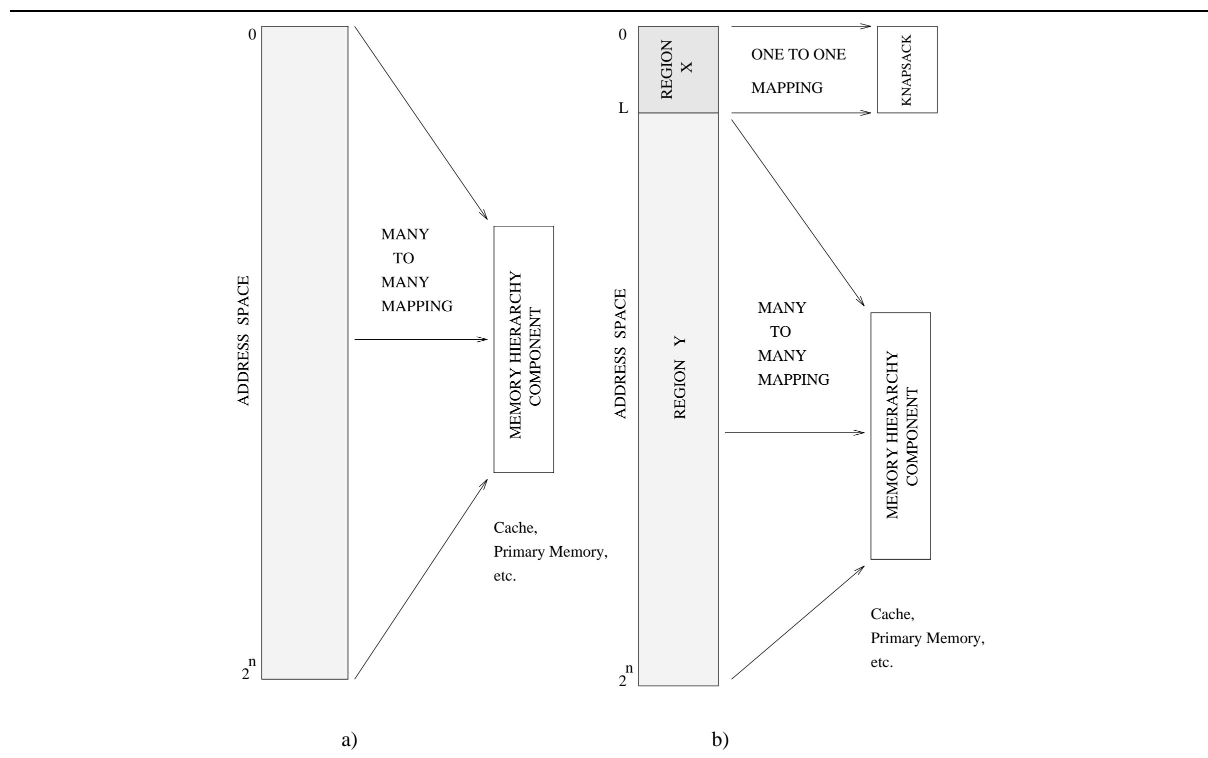


Figure 1: a) Mapping from the Address Space to the Memory Hierarchy b) Mapping from the Address Space to the Knapsack and the rest of the Memory Hierarchy

hierarchy components, for example caches, could be used to provide fast access to region Y, the *non-knapsack region*. There is a one-to-one mapping between the elements of the knapsack region and the knapsack, i.e., the size of the knapsack region is equal to the size of the knapsack.

An architecture could be completely unaware of the existence of the knapsack, that is, a knapsack is architecturally invisible: the software knows of only a single address space. Since there is only a single address space, any data object can be allocated to any part of the address space (including the knapsack region), without any concern for aliasing. An implementation is aware of the existence of a knapsack, and therefore the address range of the knapsack region. When an address is generated in an implementation, if the address is in the knapsack region, it is forwarded to the knapsack, else it is forwarded to other components of the hierarchy.

Since a knapsack is not architecturally-visible, software is not required for correctness. If an implementation has a knapsack, compilers for that implementation are instructed to place frequently-used objects in the knapsack region of the address space, in the hope that dynamically these objects would end up in the fast knapsack during an execution of the program. If software is generated without knowledge of a knapsack, it will still function *correctly*, but without the performance benefits of the knapsack. This architecture-invisibility allows a knapsack to be incorporated transparently into future implementations of existing architectures.

3.2 Knapsack Design Space

Having presented the basic concept of a knapsack, let us now consider some more important details. Before proceeding further, however, a comparison between a knapsack and a cache is in order. Both components are (typically) architecture-invisible components that map elements from the memory address space; they do not constitute a separate address space. Because of the similarities, problems in the knapsack domain have parallels in the cache domain; solutions used in the cache domain could profitably be used in the knapsack domain. However, a knapsack has a major difference: the process of mapping elements from the memory addresses space. A knapsack maps only a (very) small part of the address space, using a one-to-one mapping; a cache maps a much larger part of the address space, with a more general mapping. There are several performance implications of this restriction that will become apparent in the following discussion.

The most significant decision to be made is whether a knapsack maps virtual or physical addresses. As we shall see, mapping virtual addresses into a knapsack seem to make the most sense given our thrust, but we do not preclude mapping physical addresses. Other decisions regarding a dynamically-mapped memory hierarchy component include: (i) the mapping strategy or the placement policy, (ii) the replacement policy, (iii) what is required to make the hit decision (and what happens on a miss), and (iv) the write policy.

In a knapsack, the placement policy and the replacement policy are trivial since there is a one-to-one mapping. Making the hit decision is also quite straightforward. The hit or miss check is if the address falls into the knapsack region or not. The *Kbase ptr* which holds the address where the knapsack region starts, is used for this. See Figure 2. A reference to the knapsack consists of an address which can be viewed as an offset from the *kbase ptr*. The offset is used to index into the knapsack and a *present bit* used to indicate if the referenced word is present in the knapsack. A miss occurs if the present bit is not set. We discuss what happens on a miss, and the write policy, after we discuss which address space should be mapped into a knapsack.

Conceptually a knapsack could map either a physical or a virtual address space. A problem with mapping physical addresses is that user-level software has little control over the placement of objects in the physical address space. Compilers, which we will rely on to pack frequently-used objects into the knapsack region, can only control objects in the virtual address space. One could ask the OS to restrict page placement so that the knapsack region in a virtual space is mapped onto a knapsack region in a physical space. However, this is likely to cause heavy thrashing with multiple processes since heavily-accessed regions of many virtual spaces are mapped into the same physical space. Because of these issues, we do not consider mapping physical addresses in

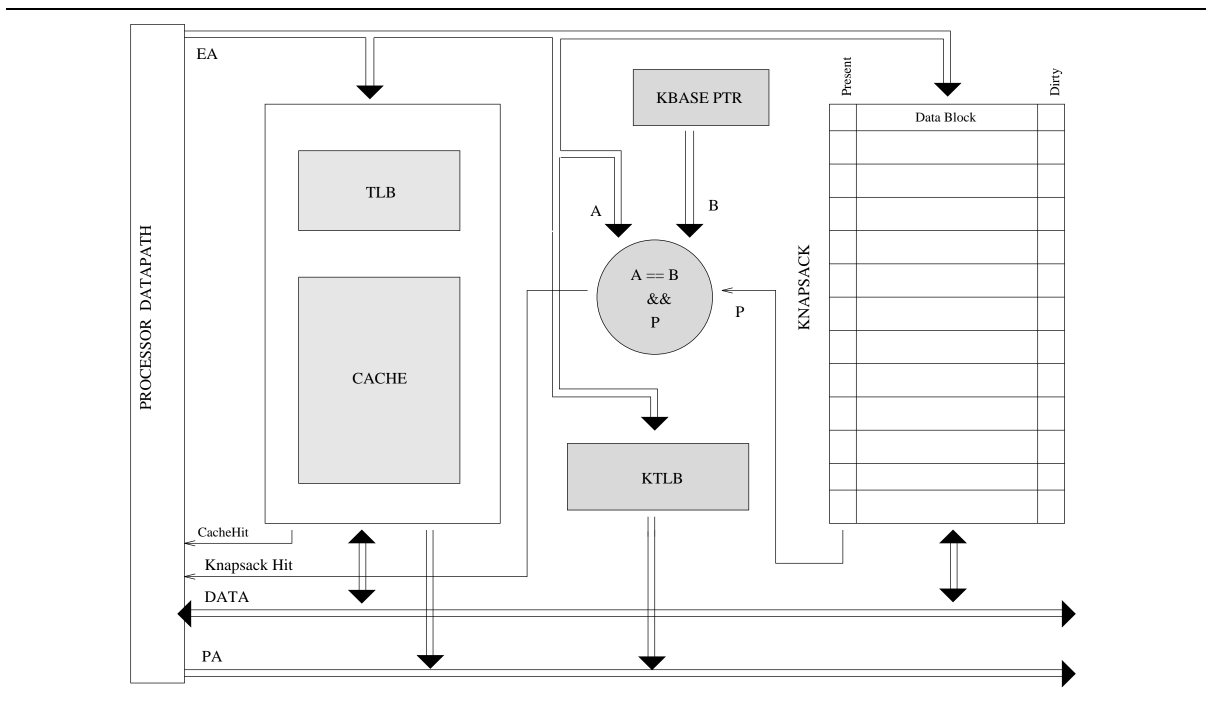


Figure 2: The Knapsack's Placement in a Conventional Memory Hierarchy.

this paper. Rather, we concentrate on mapping virtual addresses, and on solving the problems that arise when virtual addresses are mapped into a memory component. (The fundamental concepts will not change, but we will require the cooperation of another piece of software – the OS.) An important point to keep in mind for the following discussion is that a knapsack is going to contain a small number of virtual pages, contiguous in the address space, from a single address space (we discuss multiple virtual address spaces below)[4]. This restriction simplifies many of the problems associated with mapping virtual addresses. The issues to be considered here are: (1) what to do on a miss, (2) the issue of synonyms[14], and (3) the write policy.

On a miss, a translation needs to be done to retrieve the data from the physical memory. A TLB at the back end of the knapsack (called the KTLB) is needed for this purpose (see Fig. 2). If the knapsack contains only a single page, the KTLB need include only a single entry; address translation can be a simple address concatenation.

We now consider the issue of synonyms. The first concern here is whether the knapsack can contain data from multiple (virtual) address spaces at the same time, with Process Identifiers (PIDs) being used to distinguish the address space. (PIDs are commonly used to avoid flushing virtually addressed caches on an address space change.) We feel that mapping multiple virtual address spaces into a knapsack is not a good idea for two reasons. One, because of the many-

[4]In fact, if the architecture supports multiple page sizes, the compiler could instruct the OS to map all of the knapsack region into a single page – the knapsack would contain data from only a single page

to-one mapping (mapping from many virtual address spaces into a knapsack, as opposed to mapping from many places in a single address space into a single location), extra hardware, in the form of PID tags and matching logic, is needed. Two, since each process will be compiled to put heavily-used data in a knapsack, it is likely that a process will reference most of the knapsack when it is running, kicking out data from another context. Allowing multiple contexts in the knapsack therefore appears to be of questionable value.

Since only one address space is mapped into a knapsack, synonyms from multiple address spaces can not occur. However, it is possible that synonyms exist in the same address space. For example, in the Mach [18] operating system, the same object could get mapped to different addresses in the same process. Such synonyms can occur only for heap objects and since heap objects are not allocated in the knapsack, the synonyms will not cause a problem. If static objects create synonyms in the same address space, the compiler would have to be aware of this and hence can avoid allocating such objects to the knapsack.

A knapsack can use either a *write-through* or a *store-in* policy. Either policy will require a back-end address translation. If write-through is used, the knapsack need not be flushed on a context switch; the new process can start right away after the present bits have been cleared. With store-in, the knapsack needs to be flushed. The flushing process need not be as burdensome as it sounds since is may be possible to flush the knapsack on-the-fly, concurrently with the execution of the new context, with a small amount of additional hardware. The contiguous property of the data in the knapsack simplifies the flushing process since only one address translation is needed per page; (physical) addresses within a page can be formed with a simple bit concatenation.

3.3 Zero-Cycle Knapsack Access

The latency of a memory referencing operation is typically measured as the number of cycles (or pipeline stages) from the time the effective address is calculated until time the results of the memory operation are available for use. In a typical 5-stage pipeline[7] (Instruction fetch(IF), Instruction Decode(ID), Execute(EX), Memory Access(MEM), Write Back(WB)), the effective address is computed in the EX stage, and the result of a load is available at the end of the MEM stage. If the result of a load is available *before* the load enters the EX stage, we will say that it has a *zero cycle latency*. Similarly, we say that a store has a zero-cycle latency if it can be completed by the end of the EX stage.

We now see how, with suitable support from the software, a knapsack opens up the possibility of zero-cycle loads. Accesses that refer to objects allocated in the knapsack region directly by

name (and *not* through pointers) could be specified as a displacement from a *knapsack pointer*, or *kp*[5] We call such loads as *direct loads* and such stores as *direct stores.*

If software reserves a general purpose register as the *kp*, and the knapsack size is smaller than the region of memory than can be accessed with the displacement, then zero-cycle loads can be achieved in the following manner. When an instruction enters the decode (ID) stage, the displacement field bits could be used to index into the knapsack and read the contents of the accessed location. In parallel, the decode hardware can check to see if the instruction is a load, and if the base register is *kp*. In case of a hit in the knapsack, the result of the *direct load* is available at the end of the ID stage – even before the effective address (for a normal memory operation) is calculated![6]

Direct stores into the knapsack region would have to wait until it is known that the knapsack region is being accessed. This can be done in the ID stage (store address is an offset from the *kp*), and the store could complete in the EX stage. Memory operations falling into the non-knapsack region would be serviced in the MEM stage (by a cache perhaps). Memory operations accessing the knapsack region indirectly, i.e., that are *not* specified directly as displacements from the *kp*, can't be handled early in the pipeline since the effective address is calculated in the EX stage. Such references would have to be serviced, by the knapsack, in the MEM stage.

Since memory operations to the same hierarchy component (the knapsack) can complete in different stages of the pipeline, special care needs to be taken to ensure that the out-of-order memory operations do not violate dependencies. Interlocks to prevent this (which are necessary whenever there is the possibility of simultaneous or out-of-order memory operations), can be implemented with fairly routine hardware.

4 Allocating Data Objects in the Knapsack Region

We now consider the role of the software, a compiler, in making effective use of a knapsack. A key point to remember in the following discussion is that a knapsack *will function correctly without compiler help*, but likely with no performance benefits. The goal of the compiler, then, is to allocate frequently-used static objects in the knapsack region, so that they would end up in the (fast) knapsack during program execution. We can't address the multitude of issues involved

[5]It is important to realize than *not all* addresses that fall into the knapsack region of the address space could (or need) be specified directly, as a displacement from the *kp* in a general case, since this implies that we have perfect knowledge about the reference patterns of the program, and that there is no aliasing. As mentioned earlier, this is very hard, or even impossible, in the presence of control flow and pointers. Moreover, addresses to elements of arrays cannot generally be specified as a (constant) offset from a (constant) base pointer.

[6]Assuming, of course, that the memory array for the knapsack can be accessed in a single cycle.

in compiling for a knapsack, so we give only a brief overview in this paper.

A first cut for the compiler is to allocate only global objects (scalars as well as aggregates) in the knapsack region. However, we expect many frequently-used objects to be local variables of functions – variables that typically exist on a run-time stack; fast access to such objects is clearly desirable. Furthermore, if it is clear that two functions do not have overlapping lifetimes, then we can reuse the same portion of the knapsack region to hold the local objects of both functions. (We call this *temporal reuse.*) With these objectives, the compiler allocation problem can be stated as follows: allocate program objects in the knapsack region such that: (i) the knapsack is allocated to the most frequently accessed variables, and (ii) local objects from functions with overlapping lifetimes do not interfere in the knapsack.

4.1 Information about Objects

Given the above objectives, the information needed by the compiler includes: (i) the sizes of objects, (ii) the frequency of access of objects, and (iii) for local objects, information about which other functions have overlapping lifetimes.

The information about (static) object size can be obtained trivially in any compiler. Access frequency information can either be estimated statically, by analyzing the program, or be obtained using profiling. For gathering information about which functions (and therefore their local objects) do not have overlapping lifetimes, we need the program call graph. We also need the call graph to determine which functions are involved in cycles, since the number of copies of local variables for such functions is unknown at compile time, and therefore they cannot be allocated in the knapsack region.

It is quite straightforward to build the call graph for programs without calls through pointers. For programs with calls through pointers, there is a range of solutions from the most conservative (requiring no analysis) to the most accurate (requiring a lot of analysis). The simplest solution is to assume that a call through a pointer could go to any function in the program. The most accurate is to do data flow analysis[1] and converge to the least subset of functions that a call through a pointer could go to. A solution which is not too conservative but does not require a lot of analysis is to assume that a call through a pointer can go to any function whose address occurs in an expression in the program. This requires identification of functions whose addresses are taken in the program, and this can be done in a straightforward manner.

4.2 Allocation Decision

Given the relevant information, the problem of deciding what to allocate in a knapsack can be formulated as an optimization problem. The details of the formulation are beyond the scope of this paper; they will appear in a separate paper. However, it is worth mentioning that the problem resembles the well-known *fractional Knapsack Problem*. Due to the architectural invisibility of the knapsack, the compiler can allocate a variable that does not completely fit into the knapsack and let it extend across the knapsack boundary. This has an important implication on the complexity of the allocation problem.

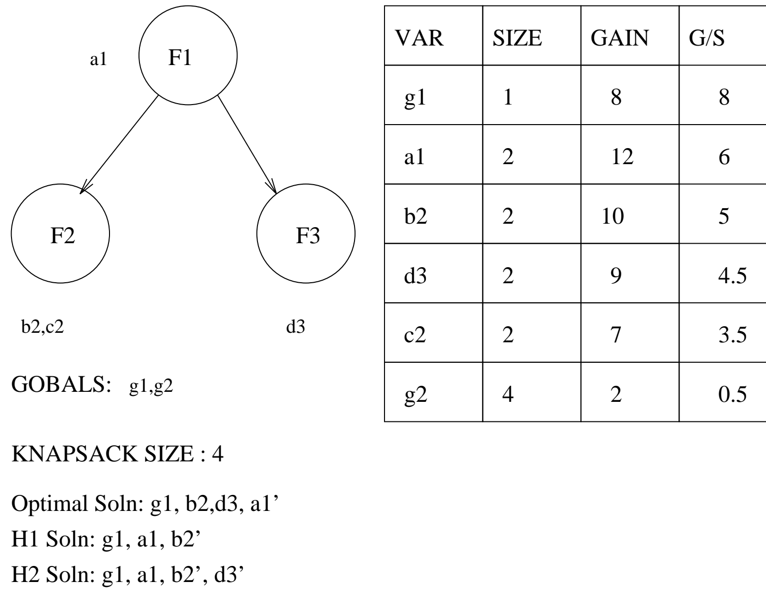
Since optimal solutions to the problem are hard, we resort to heuristics. The heuristics arrange the program objects in the order of gain per unit size, and walk through the sorted list to make the allocation decision. (With temporal reuse, the heuristic also needs to determine that a local variable object does not violate lifetime constraints in the call graph.) Since we are interested in improving memory hierarchy performance, gain for our purposes is measured as number of references.

Figure 3 gives an example of the allocation decision process. Function F1 has one local variable a1, F2 has 2 local objects b2 and c2 and F3 has one local variable d3. F1 calls F2 and F3, and F2 and F3 do not have overlapping lifetimes. There are 2 global objects g1 and g2. The size, the gain function, and the gain per unit size for each of the objects is also given in the figure. The knapsack size is 4 units. An optimal allocation will allocate g1, b2, d3 and 1 unit of a3 (denoted as a3') (d3 can use the same storage location as b2 or c2 since it is never live at the same time as b2 or c2), for a total gain of 33. A heuristic which allocates objects using gain per unit size as the metric, and does no temporal use (heuristic H1), will allocate g1, a1 and 1 unit of b2 (denoted as b2') for a total gain of 25. A heuristic with the same metric, but with temporal reuse (heuristic H2), will allocate g1, a1, (b2',d3'), for a total gain of 29.5.

Before proceeding further, it is worth mentioning that the heuristics that we have developed so far to assist us in the knapsack region allocation process are by no means the best possible. We are investigating other heuristics that can results in a better allocation decision.

4.3 Facilitating Zero-cycle Knapsack Access

Having made the allocation decision, the compiler could facilitate zero-cycle access (as described in section 3.3), by referring to knapsack-allocated objects directly as offsets from the knapsack pointer wherever it can. This requires little additional effort on the part of the compiler.



VAR	SIZE	GAIN	G/S
g1	1	8	8
a1	2	12	6
b2	2	10	5
d3	2	9	4.5
c2	2	7	3.5
g2	4	2	0.5

Figure 3: Example of the various allocation strategies.

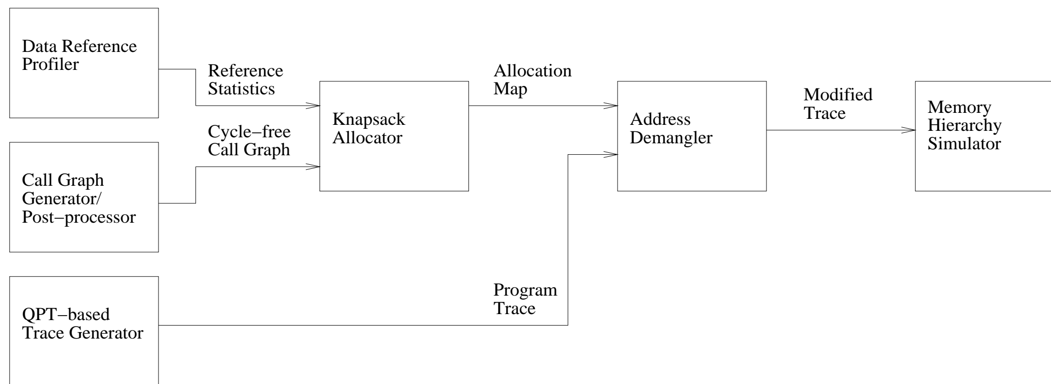


Figure 4: Experimental Framework.

5 Experimental Evaluation

5.1 Evaluation Methodology

To study the performance of the knapsack, we built a set of tools to perform knapsack allocation and simulate the memory hierarchy. Figure 4 shows our experimental framework for this work.

As discussed earlier, effective knapsack allocation requires that frequently referenced variables be allocated to the knapsack region. The data profiler provides reference frequency information for all global and local variables allocated to memory. We use QPT[12] to generate an address and call/return trace of the program being analyzed. The profiler then tries to bind each reference to a program variable name. References to the heap and to unnamed locals

(*e.g.*, temporary storage used for register spilling) cannot be bound to a program name, so we do not consider these variables for knapsack allocation. This limitation makes our simulation results somewhat conservative, as sufficient compiler and/or run-time support could allow these unnamed variables to reside in the knapsack.

To implement temporal reuse of local variables, the knapsack allocator must determine which locals have overlapping lifetimes. The program call graph provides a representation suitable for this analysis. Any two locals residing in functions that share a path in the program call graph have overlapping lifetimes. We modified GCC[17] to output function definitions, calls, calls through pointers, and function casts. We then construct the call graph by connecting all callers to callees; we conservatively approximate run-time resolved calls by connecting all calls through pointers to functions that were used in a function cast. This procedure produces a static representation of the dynamic call graph. In a post-pass, we generate the transitive closure graph of the call graph. In this graph, any function connected to itself is the member of a cycle, thus its local variables are not subject to knapsack allocation. The cycle nodes are collapsed into a single node with no locals.[7]

The knapsack allocator takes the variable reference statistics from the data profiler and the cycle-free call graph information from the call graph analyzer and produces an allocation map for the knapsack. Variables are packed in descending references-per-byte order using a packing heuristic. The packing heuristic allows locals with non-overlapping lifetimes to (possibly) reside in the same memory location. We assume the target architecture implements a traditional memory hierarchy immediately after the knapsack region, thus the last allocated variable may span the boundary between the knapsack region and the non-knapsack region.

The final stage of knapsack allocation is address demangling. We do not actually re-compile and re-link the program, instead, at simulation time we use the knapsack allocation map to generate adjusted addresses for all references. Knapsack references are adjusted to their location in the knapsack region (specified in the allocation map), and non-knapsack region references are adjusted by first subtracting out a delta equal to the total size of the preceding knapsack variables and then translating the variable to the address space immediately following the knapsack region. Demangling allows us to examine the exact address stream of the simulated system without having to rework all the compilation tools.

The memory hierarchy simulator models a memory hierarchy like that of Figure 1(b). The

[7]Again we are erring to the conservative here, as with sufficient compiler integration, some recursive locals could be allocated to the knapsack region. If the life time of a local falls completely before any potentially recursive call or completely after, the variable will never interfere with itself and it could be allocated to the knapsack region.

Program	Language	Class	Input		Analyzed Trace Length	
			Profiled	Analyzed	Insts	Refs
espresso	C	integer	bca.in	opa.in	134,680,083	31,572,136
xlisp	C	integer	fibonacci.lsp	queens-6.lsp	55,231,503	20,100,599
eqntott	C	integer	int_pri_3.eqn	short.eqn	170,571,688	38,911,048
compress	C	integer	compress.qpt	in	88,288,154	23,082,124
sc	C	integer	loada2	loada1	230,284,664	57,262,806
gcc	C	integer	1insn-recog.i	1stmt.i	143,227,311	45,784,882
doduc	FORTRAN	float	doducin (35s)	doducin (37s)	310,814,795	116,850,911
mdljdp2	FORTRAN	float	mdlj2.dat (30s)	mdlj2.dat (50s)	167,487,751	70,398,607
tomcatv	FORTRAN	float	N=33	N=65	156,387,135	53,715,341
ora	FORTRAN	float	ITER=25000	ITER=15200	199,079,533	57,583,559
alvinn	C	float	NUM_EPOCHS=2	NUM_EPOCHS=3	161,461,025	42,977,946
ear	C	float	short.m22	short.m22	508,367,705	123,172,020
mdljsp2	FORTRAN	float	mdlj2.dat (200s)	mdlj2.dat (100s)	614,279,710	152,384,363
swm256	FORTRAN	float	ITMAX=36	ITMAX=12	175,060,447	43,976,687
su2cor	FORTRAN	float	ITER=1	ITER=2	821,702,244	379,781,984

Table 2: Analyzed Programs.

knapsack region size and write policy can be specified. The non-knapsack region is modelled as a traditional memory hierarchy; at the highest level is a cache with parameterizable set size, block size, and associativity. Below the cache is a memory system with an adjustable bus width, memory access delay, and interleaving factor. The knapsack and cache writeback accesses are serialized, so if timing simulations are being performed, we can examine the effects of conflicts at the memory system.

5.2 Benchmarks

We analyzed fifteen programs from the SPEC '92 benchmark suite[16]. Figure 2 details the programs analyzed, their inputs and trace lengths. Whenever possible, we used an input other than the analyzed input to generate profile statistics. In some cases, this was not possible, as the program was not supplied with an alternate input in which case we varied a key parameter such as the number of iterations executed or the tolerated error.

All programs were compiled and simulated on DECstation 5000/3100 workstations using MIPs *cc* (version 2.1) at optimization level '-O'. The FORTRAN programs were first converted to C code using *f2c* (version 26.90).

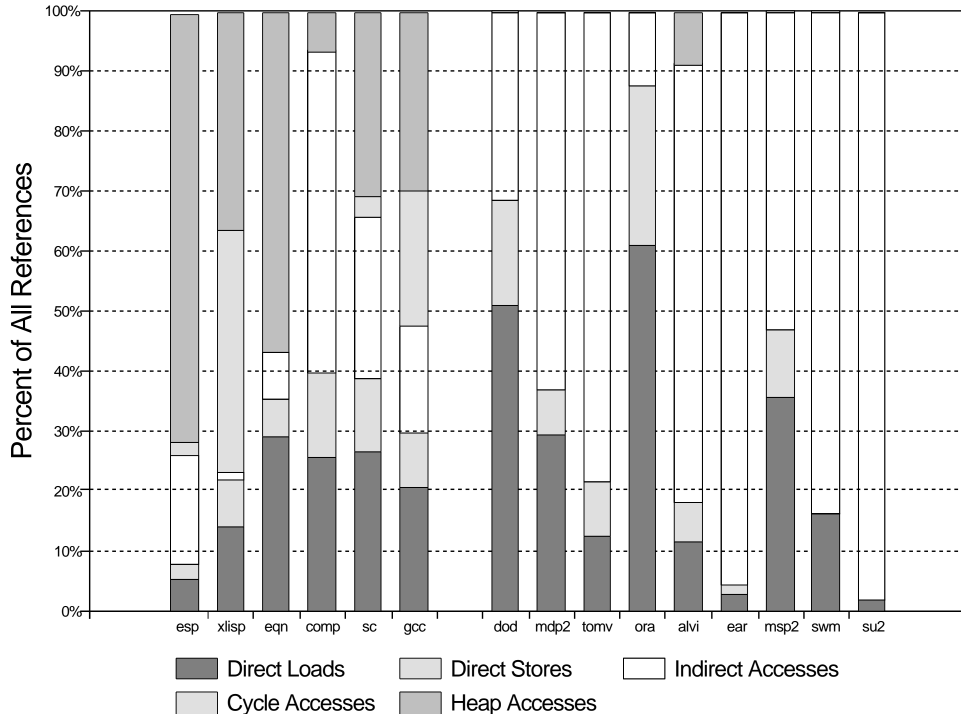


Figure 5: Reference Breakdown for All Program References.

5.3 Experimental Results

5.3.1 Knapsack Allocation

Before examining the efficacy of allocation and execution with a fixed size knapsack, we first examine the how well we could utilize an infinitely large one. With an unlimited size knapsack, the program performance improvement is simply limited by what we cannot allocate to the knapsack.

Figure 5 shows the dynamic breakdown of *all* memory references into five categories. *Direct Loads* and *Direct Stores* are memory accesses in which the address of the load or store was completely known at compile time; these accesses are typically to scalars and structures. The only array accesses that will fall under this category are those with a constant index. The *Indirect Accesses* are loads and stores to variables in which the access was indexed (*e.g.*, array accesses with a variable index) or made through a pointer. *Cycle Accesses* are accesses to variables in the local frames of (potentially) recursive functions, and the *Heap Accesses* are un-named, dynamic storage accesses. Under our allocation scheme, heap references and references to the locals of recursive functions cannot be allocated in the knapsack.

Program	Program Variable Sizes Total (% of allocatable dynamic refs)				Allocation Size	
	Globals		Locals		Naive	w/Reuse
	Scalars	Aggregates	Scalars	Aggregates		
espresso	148 (0+%)	7,410 (77%)	1,220 (23%)	372 (0+%)	9,150	7,946
xlisp	332 (99%)	6,305 (0+%)	64 (0+%)	356 (0+%)	7,057	7,029
eqntott	164 (95%)	9,618 (4%)	121 (0+%)	300 (0+%)	10,203	10,102
compress	108 (29%)	414,764 (61%)	56 (10%)	0 (0%)	414,928	414,928
sc	648 (34%)	35848 (56%)	428 (10%)	2368 (0+%)	39,292	39,012
gcc	1,766 (43%)	40,444 (25%)	1,993 (26%)	2,921 (6%)	47,124	44,523
doduc	3,328 (26%)	99,948 (48%)	2,552 (20%)	15,488 (6%)	121,316	111,920
mdljdp2	844 (36%)	204,207 (61%)	384 (2%)	368 (0+%)	205,803	205,371
tomcatv	180 (0+%)	12,212 (6%)	108 (11%)	236,600 (82%)	249,100	249,100
ora	556 (75%)	20,744 (13%)	24 (12%)	0 (0%)	21,324	21,324
alvinn	40 (2%)	456,124 (74%)	68 (0+%)	528 (24%)	456,760	456,696
ear	236 (1%)	19,992 (88%)	152 (0+%)	520 (11%)	20,900	20,748
mdljsp2	624 (19%)	162467 (76%)	124 (5%)	356 (0+%)	163,571	163,391
swm256	304 (0+%)	3,714,772 (99%)	100 (0+%)	0 (0%)	3,715,176	3,715,136
su2cor	428 (1%)	1,183,459 (29%)	752 (0+%)	1,967,020 (70%)	3,151,659	2,973,675

Table 3: Program Intrinsics.

Table 3 further decomposes the *allocatable* references into the size of the accessed storage (broadly, scalar or aggregate) and the location, either global or local. For each storage class, the table shows the total size of the class and (in parenthesis) the total percent of allocatable dynamic references directed to that particular storage class.

Since FORTRAN does not allow recursion or dynamic storage allocation, all the references made in these programs could be directed to the knapsack (given that the knapsack was large enough). The C programs, on the other hand, can employ dynamic allocation and recursion. *Xlisp* is the least amenable to knapsack allocation with 78% of its references accessing heap storage or local variables in recursive functions. Surprisingly, *gcc*, which relies heavily upon dynamic storage and recursion still allows about half of its memory references to be knapsack allocatable. *Alvinn*, *ear* and *compress* are C programs which spend most of their execution manipulating large global arrays, so most of their references are knapsack allocatable.

Opportunities for direct loads and stores vary widely. For programs which spend much of their time manipulating named scalars, *i.e.*, *compress*, *sc*, *doduc*, *mdljdp2*, *ora* and *mdljsp2*, a large fraction (37% - 88%) of the dynamic reference stream are direct accesses. *Tomcatv*, *alvinn*, *ear*, *swm256* and *su2cor* all manipulate very large array variables, thus they have few direct accesses. Any speedup realized for these programs will have to be attributed to speedup of knapsack indirect accesses or increased bandwidth through simultaneous access to the knapsack and cache.

The cost of directing references to the knapsack region is the storage required to hold the variables. Table 3 shows the amount of storage required to hold all allocatable program variables.

The *Naive* column is the total storage required if temporal reuse is not employed; *w/Reuse* is the total storage required if the variables are packed with our packing heuristic.

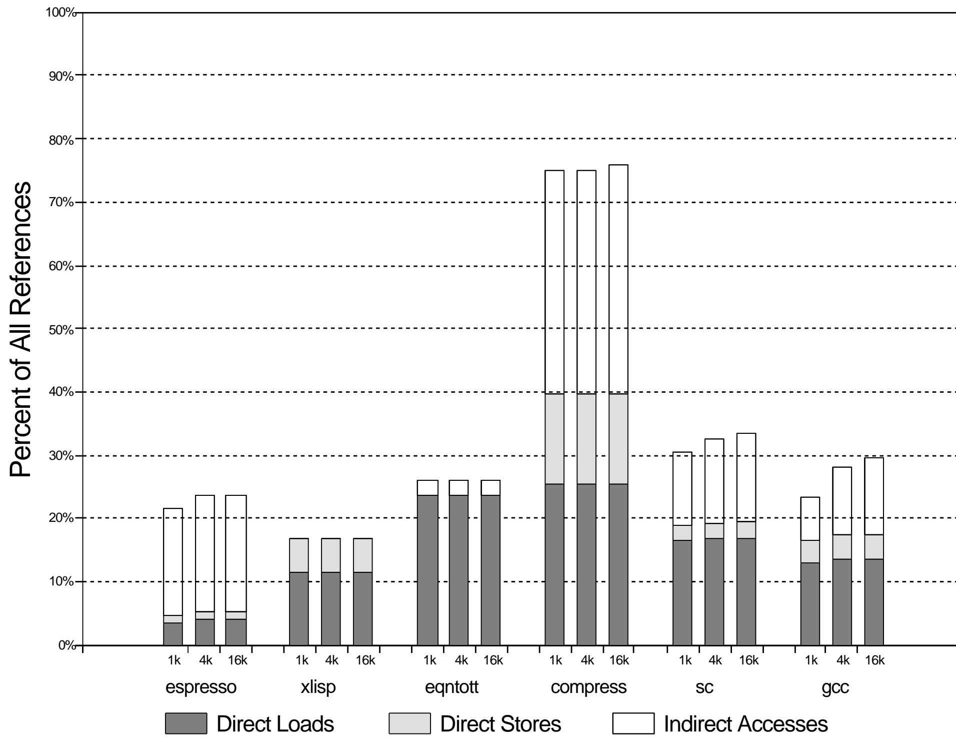
For the programs with sizable locals, *i.e.*, *espresso*, *gcc*, *doduc*, and *su2cor*, temporal reuse is quite effective for reducing the size of storage required to hold the knapsack variables. *Tomcatv* has only one function, so temporal reuse is not possible.

Having examined allocation for an infinite size knapsack, we now examine the utility of a fixed size knapsack. Figure 6 shows the reference breakdown for all references for knapsack sizes of $1k$, $4k$, and $16k$ bytes. For the non-numeric programs, a $1k$ knapsack captures nearly all of the allocatable references. A $4k$ knapsack captures slightly more references for *espresso*, *sc*, and *gcc*, and a $16k$ knapsack only marginally improves the results of *compress*, *sc* and *gcc*. The diminishing return for these program results from frequency based allocation; most of the high profit variables find their way into the knapsack very early on. For the non-numeric programs, these are typically small scalar variables; for example, for *xlisp*, 99% of the allocatable references go to 332 bytes of scalar variables.

All the numeric programs, except *ear* and *doduc* (somewhat), manipulate large array variables. As we allocate to larger knapsacks, more of the references are captured. The slope of this improvement is proportional to the size of the variables in the program, as can be seen by comparing *alvinn* and *ear*. *Alvinn* manipulates extremely large arrays, thus allocating a small portion of those arrays shows little improvement. *Ear*, on the other hand, manipulates much smaller arrays resulting in many more indirect accesses being directed to the knapsack. Without the ability to span the last allocated variable across the knapsack/non-knapsack boundary, *mdljdp2*, *alvinn*, *ear*, *mdljsp2*, and *su2cor* would show virtually no improvement since their most frequently accessed arrays are all larger than $16k$ bytes (and thus would never be allocated in the knapsack).

Table 4 shows the fraction of total references directed to the knapsack with and without temporal reuse packing, and the knapsack constituency for knapsack sizes of $1k$, $4k$, and $16k$. Also shown is the static breakdown of allocated sizes by variable size (*i.e.*, scalar vs. aggregate) and by variable location (*i.e.*, global vs. local). The number in parenthesis is the fraction of total storage from that class allocated to the knapsack.

The difference in dynamic fractions between the allocation with reuse and the allocation without reuse becomes less pronounced as the size of the knapsack increases. This is expected because if the size goes to infinity, then there is no difference between the two allocations. *Gcc* and *doduc* show a significant difference for $1k$ knapsack because they have a large collection of locals and hence more opportunity to make use of temporal reuse. In general, there is



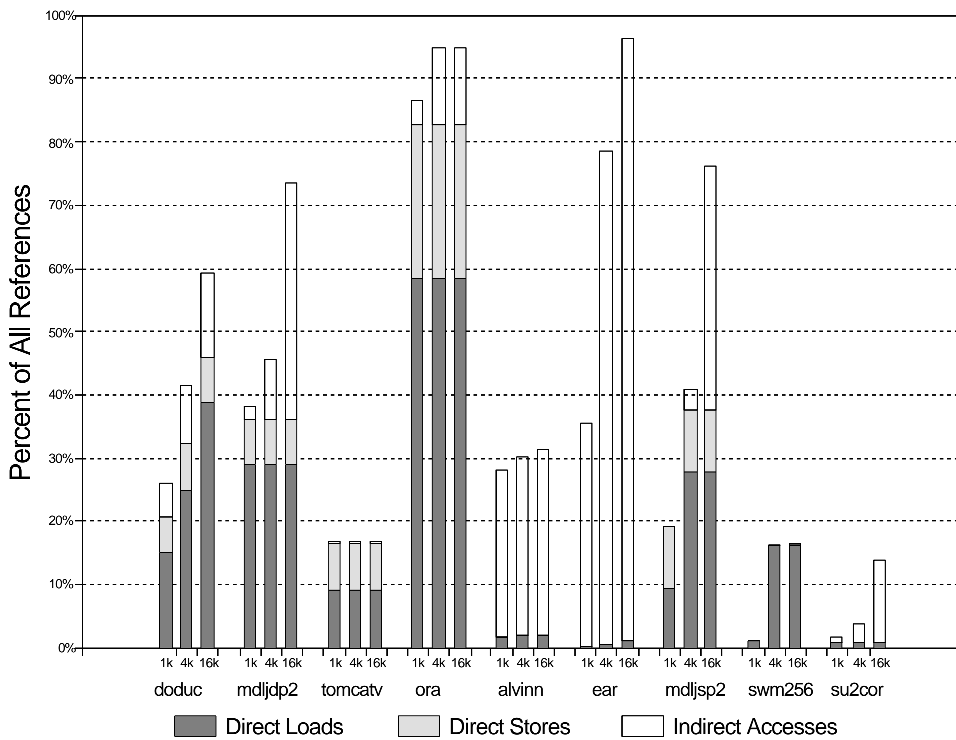


Figure 6: Reference Breakdowns for Various Knapsack Sizes.

Program	Knap-sack Size	With Temporal Reuse				Dyn Frac to Knapsack with Reuse	Dyn Frac to Knapsack w/o Reuse
		Global		Local			
		Scalar	Aggr	Scalar	Aggr		
espresso	1K	3(0.08)	3(0.20)	114(0.38)	0(0.00)	0.2164	0.2058
	4K	37(1.00)	14(0.93)	303(1.00)	5(1.00)	0.2392	0.2392
	16K	37(1.00)	15(1.00)	304(1.00)	5(1.00)	0.2393	0.2393
xlisp	1K	39(0.48)	4(1.00)	8(0.50)	0(0.00)	0.1685	0.1685
	4K	82(1.00)	9(0.82)	16(1.00)	1(1.00)	0.1688	0.1688
	16K	82(1.00)	11(1.00)	16(1.00)	11(1.00)	0.1689	0.1689
eqntott	1K	23(0.54)	7(0.35)	29(0.93)	0(0.00)	0.2629	0.2629
	4K	27(0.64)	15(0.75)	31(1.00)	1(1.00)	0.2630	0.2630
	16K	42(1.00)	20(1.00)	31(1.00)	1(1.00)	0.2631	0.2631
compress	1K	11(0.41)	5(0.50)	6(1.00)	0(----)	0.7506	0.7506
	4K	11(0.41)	5(0.50)	6(1.00)	0(----)	0.7521	0.7521
	16K	11(0.41)	5(0.50)	6(1.00)	0(----)	0.7589	0.7589
sc	1K	62(0.40)	10(0.17)	63(0.62)	0(0.00)	0.3046	0.3042
	4K	90(0.58)	23(0.40)	79(0.77)	0(0.00)	0.3289	0.3264
	16K	97(0.63)	33(0.57)	84(0.82)	2(0.50)	0.3350	0.3350
gcc	1K	105(0.24)	8(0.10)	427(0.88)	12(0.35)	0.2347	0.2103
	4K	202(0.47)	18(0.38)	448(0.92)	27(0.79)	0.2796	0.2796
	16K	252(0.58)	40(0.52)	457(0.94)	28(0.82)	0.2987	0.2963
doduc	1K	82(0.18)	4(0.02)	229(0.66)	12(0.27)	0.2604	0.1926
	4K	286(0.62)	12(0.05)	326(0.94)	19(0.42)	0.4148	0.3780
	16K	287(0.62)	25(0.11)	329(0.95)	19(0.42)	0.5948	0.5760
mdljdp2	1K	16(0.12)	1(0.01)	58(0.94)	4(0.40)	0.3824	0.3813
	4K	16(0.12)	1(0.01)	58(0.94)	4(0.40)	0.4573	0.4562
	16K	16(0.12)	5(0.06)	58(0.94)	4(0.40)	0.7350	0.7339
tomcatv	1K	17(0.43)	5(0.23)	12(0.67)	0(0.00)	0.1684	0.1684
	4K	31(0.78)	7(0.32)	18(1.00)	0(0.00)	0.1689	0.1689
	16K	40(1.00)	22(1.00)	18(1.00)	0(0.00)	0.1689	0.1689
ora	1K	28(0.29)	2(0.07)	3(0.75)	0(----)	0.8663	0.8663
	4K	28(0.29)	3(0.10)	3(0.75)	0(----)	0.9508	0.9508
	16K	76(0.79)	15(0.52)	3(0.75)	0(----)	0.9508	0.9508
alvinn	1K	2(0.20)	2(1.00)	17(1.00)	3(0.75)	0.2798	0.2798
	4K	2(0.20)	9(0.43)	17(1.00)	3(0.75)	0.3015	0.3014
	16K	2(0.20)	11(0.52)	17(1.00)	3(0.75)	0.3148	0.3148
ear	1K	1(0.01)	2(0.06)	0(0.00)	0(0.00)	0.3565	0.3565
	4K	7(0.12)	7(0.23)	30(1.00)	1(1.00)	0.7874	0.7872
	16K	57(0.98)	28(0.90)	30(1.00)	1(1.00)	0.9654	0.9654
mdljsp2	1K	11(0.09)	1(0.01)	17(0.59)	1(0.10)	0.1936	0.1920
	4K	11(0.09)	2(0.03)	17(0.59)	1(0.10)	0.4092	0.4091
	16K	15(0.11)	8(0.10)	17(0.59)	1(0.10)	0.7617	0.7617
swm256	1K	4(0.06)	1(0.02)	10(0.40)	0(0.00)	0.0092	0.0092
	4K	12(0.17)	2(0.04)	10(0.40)	0(0.00)	0.1600	0.1600
	16K	12(0.17)	2(0.04)	10(0.40)	0(0.00)	0.1650	0.1650
su2cor	1K	3(0.03)	2(0.03)	52(0.35)	5(0.05)	0.0160	0.0158
	4K	3(0.03)	2(0.03)	108(0.69)	12(0.12)	0.0372	0.0359
	16K	3(0.03)	2(0.03)	108(0.69)	22(0.22)	0.1381	0.1104

Table 4: Knapsack Constituency.

no significant difference between the two allocations for a knapsack of size $4k$ or more. A large number of global scalars get allocated to the knapsack. Traditional register allocation, due to aliases and relatively small number of registers, does not allocate registers to so many global scalars. In comparison to caches, direct accesses to these globals would cut down the access latency more. Global aggregates get into the knapsack if they are small enough. In the FORTRAN programs, the global aggregates are mostly too big to fit into even a $16k$ knapsack. Almost all of the locals fit into even a $1k$ knapsack. There are not many local aggregates used in the programs. But, whatever few local aggregates are present get allocated in knapsack of size $4k$ or more. Only for *su2cor* and *doduc*, many of the local aggregates are too big to fit even in a $16k$ knapsack. Overall, for the C programs a knapsack of size $1k$ comes close to accommodating as many variables as a $4k$ knapsack. There is practically no difference between $4k$ and $16k$ knapsacks. On the other hand, the FORTRAN programs have much larger global and local aggregates and hence the larger knapsacks consistently allocate more than the smaller ones.

5.3.2 Impact on Cache Performance

Introduction of the knapsack memory component into the memory hierarchy has the effect of diverting a portion of the reference stream away from the cache to the knapsack. Table 5 shows the cache hit rates for a memory system with a $4k$ byte knapsack memory and either $16k$, $32k$, $64k$, or $128k$ byte direct-mapped cache memories (with 32 byte blocks). Also shown (in parenthesis) is the cache hit rate for the same execution without the $4k$ knapsack memory. For each simulation, we also incorporated the effects of context switching by flushing the cache each $500,000$ instructions and at each system call.

We expected that the introduction of the knapsack would adversely effect the cache hit rate of the remaining reference stream. This suspicion arose because we were diverting from the cache reference stream the most frequently accesses variables, those with very high temporal locality. In all cases, except *compress*, the difference in the cache hit rate when executing with a $4k$ knapsack is almost negligible, even for a $16k$ direct-mapped cache. With larger caches, the difference is even less noticeable.

Compress, on the other hand, spends nearly all of its execution manipulating a few small scalars and a very large hash table. Knapsack allocation pulls all of the scalar accesses out of the cache reference stream leaving only accesses to the hash table array. These hash table array accesses are very sparse and exhibit little spatial or temporal locality. The resulting cache performance is very poor.

One program, *su2cor*, showed an improvement in cache performance for $32k$ and $64k$ cache

Program	Cache Size Hit Rate with 4k knapsack (hit rate without knapsack)			
	16k	32k	64k	128k
espresso	0.9798 (0.9823)	0.9890 (0.9900)	0.9919 (0.9935)	0.9924 (0.9939)
xlisp	0.9591 (0.9664)	0.9672 (0.9717)	0.9709 (0.9750)	0.9731 (0.9752)
eqntott	0.9495 (0.9600)	0.9560 (0.9650)	0.9616 (0.9709)	0.9666 (0.9750)
compress	0.4197 (0.8495)	0.4702 (0.8642)	0.5318 (0.8811)	0.5997 (0.8988)
sc	0.8119 (0.8467)	0.8133 (0.8482)	0.8138 (0.8488)	0.8143 (0.8511)
gcc	0.9103 (0.9230)	0.9538 (0.9569)	0.9637 (0.9684)	0.9679 (0.9748)
doduc	0.9568 (0.9639)	0.9673 (0.9757)	0.9771 (0.9875)	0.9802 (0.9880)
mdljdp2	0.9788 (0.9838)	0.9831 (0.9884)	0.9880 (0.9913)	0.9888 (0.9935)
tomcatv	0.9116 (0.9250)	0.9132 (0.9271)	0.9570 (0.9638)	0.9605 (0.9669)
ora	0.9992 (0.9996)	0.9992 (0.9996)	0.9992 (0.9997)	0.9992 (0.9997)
alvinn	0.9575 (0.9672)	0.9632 (0.9721)	0.9648 (0.9745)	0.9661 (0.9757)
ear	0.9942 (0.9979)	0.9944 (0.9980)	0.9946 (0.9980)	0.9948 (0.9980)
mdljsp2	0.9762 (0.9874)	0.9792 (0.9894)	0.9878 (0.9894)	0.9883 (0.9930)
swm256	0.9157 (0.9311)	0.9182 (0.9355)	0.9182 (0.9356)	0.9182 (0.9356)
su2cor	0.7709 (0.7745)	0.8634 (0.8602)	0.9046 (0.9036)	0.9276 (0.9281)

Table 5: Impact on Cache Performance.

sizes when the $4k$ knapsack was added. Closer examination of the knapsack allocation gives a possible clue as to why this effect occurs. With a $4k$ knapsack, nearly all scalars and $6,052$ bytes[8] of frequently accessed array storage are allocated to the knapsack. The array variables likely exhibited poor temporal and spatial locality in the cache, thus by removing them from cache (through knapsack allocation) we can increase the hit rate. The effect diminishes with larger caches.

6 Concluding Remarks

The performance of a processor continues to be dictated by the performance of the memory hierarchy: the increasing gap between logic and memory speeds demands novel memory hierarchies. In the age of increasing on-chip real estate, it is tempting to incorporate larger amounts of existing memory hierarchy components (e.g., caches) on a chip. We feel that a brute-force increase in the size of known memory hierarchy components may not be the best solution, and other avenues need to be explored.

This paper presented and evaluated a *knapsack* – a novel memory hierarchy component. The goal of a knapsack was to provide very fast (register-like in many cases) access to frequently-used data objects, and still be integrated transparently into an implementation of an existing architecture. A knapsack achieves its goal of fast access by mapping a restricted part of the memory

[8]This value is larger than $4k$ because some of the allocated variables were arrays with non-overlapping lifetimes.

address space (the knapsack region) into fast storage elements (the knapsack). By restricting the mapping to one-to-one (instead of the many-to-many found in a cache, for example), a knapsack simplifies the run-time decision-making process, resulting in fast access. Access is further sped up (reduced to register access times, or zero-cycle access) for memory references that can be specified as a constant offset into this knapsack region at compile time. To achieve good performance for a knapsack, however, extensive use of sophisticated software is required. We discussed the requirements of the compiler to make effective use of a knapsack.

We also carried out a detailed (but by no means complete) evaluation of a knapsack, using most of the SPEC92 benchmarks. Our evaluation showed that a significant portion of program references could be directed to the knapsack, even for programs which employ dynamic storage allocation and recursion. Using a packing heuristic that allocates both global and local variables to the knapsack, we found that a knapsack as small as $4k$ could capture most of the allocatable references; for the non-numeric programs, as little as $1k$ could. We also evaluated the impact of diverting part of the reference stream away from the cache, and saw that the impact, for most programs, was negligible.

Size limitations allowed us to address only a few of the potential performance benefits of a knapsack in this paper. We feel that a knapsack allows many performance optimizations that still need to be explored. For example, we feel that the presence of a knapsack facilitates the multiporting of the memory system because: (i) a knapsack provides a second memory port; references to the knapsack and a cache can be serviced simultaneously, and (ii) it is easier to multiport a knapsack than it is to multiport a cache. Further performance benefits of a knapsack need to be explored and evaluated. We also need to investigate more sophisticated use of a knapsack, for example: (i) allocating portions of the heap and (some) local variables of functions involved in cycles into the knapsack, and (ii) better compiler heuristics (and algorithms) for allocating variables in the knapsack region. Finally, we need to further evaluate the multitude of issues that arise in the design of any memory hierarchy component, such as write policies, which address space it should map, impact of context switches, its interactions with other hierarchy components, etc.

Acknowledgements

We would like to thank Jim Goodman, Mark Hill, and David Wood for their insightful comments on this paper.

References

[1] F. E. Allen and J. Cocke. A program data flow analysis procedure. *CACM*, 19(3):137–147, 1976.

[2] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, Santa Clara, CA, April 1991.

[3] W. Y. Chen, S. A. Mahlke, P. P. Chang, and W. W. Hwu. Data access microarchitectures for superscalar processors with compiler-assisted data prefetching. In *Proceedings of the 24th Annual Workshop on Microprogramming and Microarchitecture (Micro 24)*, pages 69–73, November 1991.

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* McGraw Hill Book Company, 1990.

[5] Cray Research, Inc., Mendota Heights, MN. *Cray Computer Systems: Cray-2 Hardware Reference Manual*, 1985.

[6] D. R. Ditzel and H. R. McLellan. Register allocation for free: the C machine stack cache. In *1st International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 48–56, Palo Alto, CA, March 1982.

[7] J. L. Hennesey and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Mateo, CA, 1990.

[8] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, Seattle, WA, May 1990.

[9] A. C. Klaiber and H. M. Levy. An architecture for software-controlled data prefetching. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 43–53, May 1991.

[10] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pages 81–87, May 1981.

[11] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of block algorithms. In *4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, Santa Clara, CA, April 1991.

[12] James R. Larus. Efficient program tracing. *IEEE Computer*, 26(5):52–61, May 1993.

[13] R. M. Russell. The CRAY-1 computer system. *CACM*, 21(1):63–72, January 1978.

[14] A. J. Smith. Cache memories. *Computing Surveys*, 14(3):473–530, September 1982.

[15] G. S. Sohi and M. Franklin. High-bandwidth data memory systems for superscalar processors. In *4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 53–62, Santa Clara, CA, April 1991.

[16] SPEC newsletter, December 1991.

[17] Richard M. Stallman. *Using and Porting GNU CC.* Free Software Foundation, 1993.

[18] Michael Young, Avidis Tevanian, Richard F. Rashid, David Golub, J. Eppinger, Jonathan Chew, William Bolosky, David Black, and Robert Baron. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proceedings of the 11th Symposium on Operating System Principles*, November 1987.