A Case for Compiler-driven Superpage Allocation

Joshua Magee
Department of Computer Science
Texas State University
San Marcos, TX
jm1576@txstate.edu

Apan Qasem
Department of Computer Science
Texas State University
San Marcos, TX
apan@txstate.edu

ABSTRACT

Most modern microprocessor-based systems provide support for superpages both at the hardware and software level. Judicious use of superpages can significantly cut down the number of TLB misses and improve overall system performance. However, indiscriminate superpage allocation results in page fragmentation and increased application footprint, which often outweigh the benefits of reduced TLB misses. Previous research has explored policies for smart allocation of superpages from an operating systems perspective. This paper presents a compiler-based strategy for automatic and profitable memory allocation via superpages. A significant advantage of a compiler-based approach is the availability of data-reuse information within an application. Our strategy employs data-locality analysis to estimate the TLB demands of a program and uses this metric to determine if the program will benefit from superpage allocation. Apart from its obvious utility in improving TLB performance, this strategy can be used to improve the effectiveness of certain data-layout transformations and can be a useful tool in benchmarking and empirical tuning. We demonstrate the effectiveness of this strategy with experiments on an Intel Core 2 Duo with a two-level TLB.

Categories and Subject Descriptors

D.3.4 [**Software**]: Processor—*compilers, memory management*

General Terms

Design, Performance, Experimentation

1. INTRODUCTION

As application data footprints grow larger, so too does the importance of the memory hierarchy in improving program performance. The translation lookaside buffer (TLB), which lies in the critical path of a memory access plays an important role in improving application performance. Studies have shown that for many data-intensive applications, increased TLB misses can not only degrade performance but in many cases becomes the principal bottleneck [7]. Because of its importance to improving application performance, the TLB has received considerable attention from industry and academia alike. Many strategies have been proposed for improving TLB performance both at the hardware and software level.

Among the different strategies proposed, the use of superpages has been the dominant one. Most micro-processor based systems today support multiple page sizes, from smaller pages of size 4K-16K to larger pages of size 2M-16M. The use of large pages increases TLB coverage and reduces TLB misses. However, indiscriminate use of large pages leads to unwarranted increase in application data footprint and causes internal page fragmentation. To ameliorate the ill-effects of fragmentation, several strategies have been proposed that use heuristics for intelligent allocation of superpages. Since the operating system is primarily responsible for allocating physical pages, most techniques for managing superpages have been OS-centric. However, because of the role the compiler plays in program analysis and in setting up the run-time environment, it can help the operating system in efficient allocation of superpages. There are two key advantages to having a compiler-based scheme for superpage allocation:

(i) Programmer productivity and code portability: On most systems, to acquire superpages, the programmer needs to insert code that sends an explicit request to the operating system through the memory allocator. Although there are APIs that can streamline the code to be inserted, the responsibility to request pages still remains with the programmer. Having automatic support for allocation into superpages at the compiler-level relieves the programmer from this responsibility. Moreover, since support for superpages vary across systems, programs that contain explicit code for superpage requests become less portable. Having the compiler insert the platform-specific code increases portability and maintainability.

(ii) Enhanced information for allocation decisions: A key advantage the compiler has over the operating system is that it has knowledge of the memory access behavior of an application. The OS, when allocating memory for a process only considers the data footprint - that is the amount of memory required by the program. However, the data footprint does not necessarily indicate how much pressure the application is going to put on the TLB. The actual TLB usage

ACMSE '09 March 19-21, 2009, Clemson, SC, USA.

for a program depends on its data-reuse patterns. Generally, the number of distinct pages touched within the *working set* determines the TLB traffic for a particular application. Thus, allocation decisions that are oblivious to the data-reuse patterns within an application are likely to be less effective. Since the *working set* information can only be derived through data-dependence analysis, the compiler can play an important role in making superpage allocation heuristics more effective.

In this paper, we present the design and implementation of a compiler-based approach for allocation of superpages that yields the advantages discussed above. Our strategy is fully automatic and uses data-reuse information within an application to make allocation decisions. We provide preliminary experimental results that show that our technique can be effective in reducing the number of TLB misses, while preserving code portability and maintainability.

2. RELATED WORK

There has been significant work in developing strategies for improving TLB performance both at the hardware and software level. Hardware approaches have mainly focused on either modifying TLB organization or extending the existing TLB architecture. Talluri and Hill propose a TLB organization based on *partial subblocks* that can extend TLB coverage with minimal support from the operating system [10]. Fang et al. propose a two-level address translation mechanism that allows placement of multiple smaller pages into a larger page [3].

Software strategies for improving TLB performance have focused on extending TLB coverage and providing transparent support for large pages. Navarro et al. provide a superpage management strategy for operating systems and demonstrate that it is effective in lowering TLB misses, increasing application performance and reducing fragmentation [8]. Shimizu et al. extend this work and provide a Linux implementation for superpages that yields significant performance improvement on the tested workloads [9].

Lu et al. propose modifications to the Linux kernel for mapping application text regions to superpages for enterprise workloads [5]. As discussed earlier our proposed strategy for compiler-based superpage allocation is complementary to any of the OS-based approaches.

3. IMPLEMENTATION

We implemented automatic support for superpages in the Low Level Virtual Machine (LLVM) Compiler Infrastructure [4]. LLVM provides a framework for language independent analysis and optimization, front-end development, and compile, run, and link-time optimizations and thus, served as an ideal platform for implementing our strategy. Providing superpage support required restructuring and enhancing several component modules within LLVM, including the runtime memory management system and the source-level transformation pass. In addition, we implemented a new API for memory allocation into superpages. In this section, we briefly describe different parts of our framework. Details of the implementation can be found in [6].

Our API for superpage-aware memory allocation is called `smalloc` and is based on algorithms used in the `malloc` implementation in `glibc`. `smalloc` supports the core interface of the C standard, including `scalloc`, `srealloc` and `sfree`. With `smalloc`, memory can be backed by either base pages, superpages, or to a limited extent both. `smalloc` requests memory by calling one of two versions of the `morecore` function (adopted from `glibc`). Base pages are obtained by `mmap`ping `/dev/zero` and superpages are obtained by `mmap`ping a file backed by the `hugeTLBfs`, provided by Linux. Each version aligns the base of the heap at the end of the data segment.

Since `smalloc` allows dynamic allocation of both superpages and base pages, the memory management system in LLVM was modified to accommodate both types of pages. Our implementation of the runtime system, allocates space for both base pages and superpages in the program heap and allows for transparent switching between the two. In addition, the system also provides *heap migration* and *base heap freezing* to reduce fragmentation within the heap [6].

Finally, we implemented a pass in LLVM to traverse the source code and replace calls to `malloc` with the corresponding calls to `smalloc` and insert the appropriate header files. We also provide command-line options in LLVM to disable superpage usage and choose any of the implemented heap management strategies. Thus, our implementation completely hides the details of superpage allocation at the application level, thereby increasing code portability.

4. A HEURISTIC FOR SELECTIVE SUPERPAGE ALLOCATION

A compiler-based strategy for superpage allocation with no heuristic for making allocation decisions is unlikely to be effective in practice. Applications vary widely in terms of their memory footprint. Even for applications with similar footprints there can be wide variations in TLB access patterns. Allocating superpages to programs with low TLB demands causes unwarranted increases in the memory footprint and leads to page fragmentation, compromising the overall performance of the system.

To successfully exploit superpages the compiler must be able to estimate the TLB demands of an application and determine if these demands will benefit from the advent of superpages. A high-level sketch of our algorithm is given in Fig. 4. The key idea behind our heuristic is to allocate superpages only when the demand for memory pages for the working set exceeds the capacity of the target TLB. Therefore, making a smart allocation decision essentially boils down to estimating two parameters: *threshold* and *spread*. The *threshold* represents a conservative estimate of the effective capacity of the TLB, whereas *spread* is an estimate of the number of distinct pages touched within the working set of an application. The *threshold* must take into account the page size and the capacity and associativity of the TLB. To estimate the *threshold*, we employ the following simple formula, based on earlier models for set-associative caches:

$$threshold = (\frac{no\ of\ entries}{associativity} + 1) \times\ base\ page\ size$$

We adopt a dependence-based approach for estimating the *spread* of an application [1]. The algorithm examines each loop nest and each array reference within the loop body. Array references are categorized in terms of the type of *data dependence* they exhibit. The number of distinct base pages touched by each reference is estimated based on the particular type of dependence associated with the reference. For

```
for all loop-nests s in procedure do
  for all memory references r in s do
    I ← number of current loop iteration
    if r depends on loop induction variable then
      spread_r ← 1
    else if r strides over non-contiguous dimensions then
      spread_r ← I
    else if r strides over contiguous dimensions then
      s ← step size
      spread_r ← (I·s) / base_page_size
    end if

    if r varies with the loop index then
      spread_r ← spread_r · I
    else
      spread_r ← spread_r · 1
    end if
  end for
end for

spread_all ← 0
for all memory references r do
  spread_all ← spread_all + spread_r
end for

if spread_all ≥ threshold then
  Use superpages
else
  Use standard pages
end if
```

Figure 1: Heuristic for Superpage Allocation

example, for a loop independent dependence, a value of 1 is assigned, whereas for a outer-loop carried dependence a value of I is assigned, where I is the current loop iteration. Once a *spread* value has been attributed to each memory reference, they are summed up to estimate the total *spread* for the working set.

5. EXPERIMENTAL EVALUATION

5.1 Experimental Setup

We evaluated our strategy on a 2.33 GHz Intel Core 2 Duo. The architecture provides a 4-way TLB with 256 entries for 4KB pages and a 4-way TLB with 32 entries for 4MB superpages. The benchmarks used in this work include *164-gzip*, *188-ammp*, *176-gcc*, *256-bzip2* and *183-equake* from SPEC 2006, *transpose*, a matrix transposition code and *stride*, a synthetic kernel that strides through a large array with increasing step sizes. The Performance Application Programming Interface (PAPI) is used to collect performance metrics. PAPI provides detailed performance measurements for a variety of metrics by sampling hardware performance counters via the *perfctr* module in the Linux kernel [2].

Each benchmark trial consists of two executions of the application with a given data set[1]. One execution is configured to run with standard pages and the other utilizes superpages. The input data for the SPEC benchmarks are obtained from the SPEC reference data sets. The input data for *transpose* is randomly generated for increasing step sizes of 3600 bytes. The results of several metrics from each test

[1]Each trial is repeated several times to account for measurement aberrations

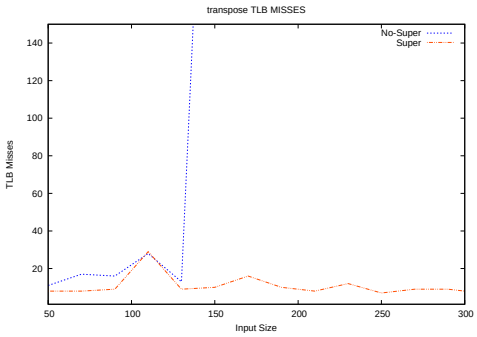

Figure 2: *transpose* TLB performance

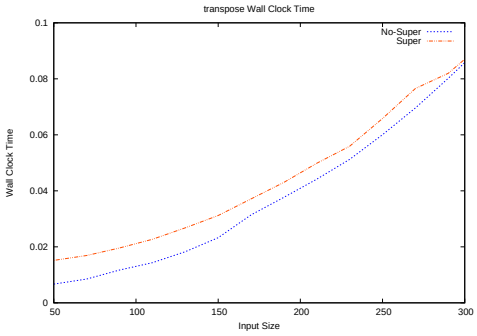

Figure 3: *transpose* Wall Clock Time

are sampled using PAPI. The metrics sampled includes TLB misses, L2 cache misses, wall clock time, and total clock cycles, with the number of TLB misses being of foremost importance. The results of the execution is recorded and pushed to a database and the same benchmark is profiled with the next data set.

5.2 Discussion

In the interest of space, we provide detailed results on two benchmarks and provide summary results for the rest. Fig. 2 shows number of TLB misses for *transpose* with and without superpages. The domain is the range of input matrix sizes from 50·50 to 300·300. The variation of TLB misses between the two page sizes is relatively small until input dimension $130 \cdot 130$, at which point the standard-sized page execution incurs a marked number of TLB misses. This marked divergence occurs at an input size of approximately 66 KB. Fig. 3 shows wall clock time for *transpose*. Over the entire domain of inputs there is an improvement to the execution time when using superpages.

Fig. 4 shows number of TLB misses for *164-gzip* for different input sizes. This benchmark is an ideal example of an application that reaps little or no benefit from the advent of superpages. *164-gzip* is characterized by linear data access patterns, using a floating window as it passes over the data. The cost of fragmentation in this benchmark is not offset by any gains in performance, and therefore superpage allocation should not be recommended.

Fig. 5 shows the average reduction of TLB misses for each benchmark. We observe that all benchmarks except for *164-gzip* and *stride*, realize significant benefits from the use of

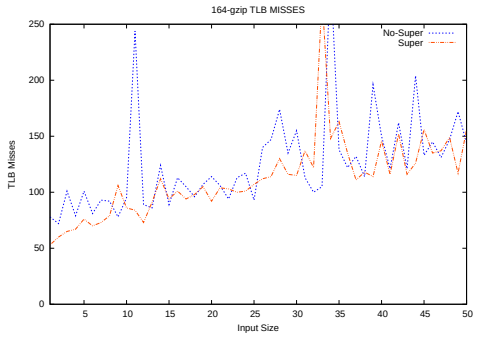

Figure 4: *164-gzip* TLB Misses

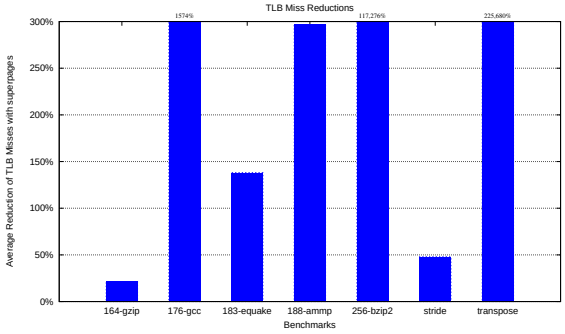

Figure 5: TLB Miss Reductions for all Benchmarks

superpages. The reduction in TLB misses for *188-ammp* and *176-gcc* is particularly remarkable. These numbers suggest severe TLB thrashing for both *188-ammp* and *176-gcc* when using base pages. The heuristic from Section 3, when applied to the benchmarks, was able to correctly predict that *164-gzip* would not benefit from superpage allocation. The heuristic did recommend that superpages be allocated for *stride*, although we do not observe a significant benefit for this benchmark. Since *stride* performs a strided access through memory, we speculate that hardware prefetching played a role in curbing some of the TLB misses. Of course, this is a limitation of our strategy which we plan to address in future (see Section 7).

6. CONCLUSION

This paper presents a technique for smart superpage allocation guided by compiler heuristics. We provide an implementation of our strategy in the LLVM compiler infrastructure and demonstrate its effectiveness through some preliminary experiments on the Intel Core 2 Duo platform. The experimental results suggest that our heuristic is able to correctly evaluate an application's need for superpage memory and advise the operating system accordingly.

7. FUTURE WORK

The work presented in this paper demonstrates that compiler guided superpage allocation can indeed be beneficial in some cases. To make a stronger case for a compiler-based approach, however, more extensive experiments will need to be performed. Our future plans include evaluating our model on a larger set of benchmarks and on platforms with different TLB configurations. We also plan to augment our heuristic to handle imperfectly nested loops and account for hardware and software prefetching.

Another future direction for this work will be to investigate how superpages can be exploited to enhance compiler optimizations. Many memory hierarchy transformations, particularly ones that aim to reduce conflict misses, are most effective when the compiler has complete knowledge of how data within a program are mapped to different cache lines. In the absence of such information, the compiler either assumes that allocation of memory is contiguous or takes a *guess* at the most likely mapping. Use of superpages guarantees contiguous memory allocation (at least for large chunks of data) and eliminates much of the guess work for the compiler, which can lead to more effective heuristics for optimizations. In the future, we plan to leverage this work to design superpage-aware heuristics for data-layout transformations such as *array padding* and *data copy*.

8. REFERENCES

[1] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 2002.

[2] S. Browne, J. Dongarra, N. Garner, G. HO, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *International Journal of High Performance Computing Applications*, 14(3), 2000.

[3] Z. Fang, L. Zhang, J. B. Carter, W. C. Hsieh, and S. A. Mckee. Reevaluating online superpage promotion with hardware support. In *In Proceedings of the Seventh International Symposium on High Performance Computer Architecture*, 2001.

[4] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Mar 2004.

[5] H. J. Lu, K. Doshi, R. Seth, and J. Tran. Using hugetlbfs for mapping application text regions. In *Proceeding of the Ottawa Linux Symposium*, 2006.

[6] J. Magee. Automated compiler driven superpage allocation and its applications. Master's thesis, Texas State University, Dec. 2008.

[7] G. Marin and J. Mellor-Crummey. Pinpointing and exploiting opportunities for enhancing data reuse. In *Proceedings of the 2008 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'08)*, 2008.

[8] J. Navarro, S. Iyer, P. Druschel, and A. Cox. Practical, transparent operating system support for superpages. *Fifth Symposium on Operating Systems Design and Implementation*, 2002.

[9] N. Shimizu and K. Takatori. A transparent linux super page kernel for alpha, sparc64 and ia32: reducing tlb misses of applications. *SIGARCH Comput. Archit. News*, 31(1):75–84, 2003.

[10] M. Talluri and M. D. Hill. Surpassing the tlb performance of superpages with less operating system support. In *ASPLOS-VI: Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1994.